XQuery

- XQuery ist eine funktionale Sprache; Anfragen werden nicht in Form von Anweisungen, sodern als ein Ausdruck definiert.
- XQuery ist eine streng-getypte Sprache.
- Der Wert eines Ausdrucks ist eine Sequenz, d.h., eine geordnete Menge von keinem oder mehreren Items.
- Ein Item ist entweder ein atomarer Wert, oder ein Knoten.
- Ein Knoten ist ein Document-, Element-, Attribut-, Text-, Name-Space-, Processing-Instruction-, oder Comment-Knoten.
- Ein Knoten hat eine Identität.

Dokument-Ordnung

- XML-Dokumente liegen als geordnete Bäume von Knoten vor.
- Die Document-Order ergibt sich aus der Preorder (Hauptreihenfolge; top-down, left-to-right) des Baumes.
  - Die Wurzel des Baumes ist der Dokument-Knoten.
  - Element-Knoten, Text-Knoten, Kommentar-Knoten, Processing-Instruction-Knoten sind in der Reihenfolge ihres Auftretens im Dokument.
  - Attribut-Knoten sind nicht Teil des Baumes, haben jedoch eine feste Position in der Document-Order.
  - Attribut-Knoten stehen nach ihrem Element-Knoten und vor dessen Kindern.

Literale und Kommentar

- Kommentar wird eingeschlossen in ”(:”, ”:)”.
- Numerische Literale sind Integer (1, +2, -2), Numeric (1.23, -1.23) und Double (1.2e5, -1.2E5).
- String-Literale sind in Hochkommata ” ” oder Apostroph ’ ’ eingeschlossen: ”a string”, ’a string’, ”it’s a string”, ’it”s a string’, ”a ”” or a ’ delimits”, ’a ” or a ” delimits’. Entity-References können enthalten sein: ”Ich bin müde.”

Lokation von Knoten: Path Expressions (XPath)

- Ein Path besteht aus einer Folge von einem oder mehreren durch / oder // getrennten Steps. Jeder Step definiert eine Menge von Knoten.
- Eine durch einen Step definierte Menge von Knoten kann durch ein mit ”[”, ”]” eingeschlossenes Prädikat eingeschränkt werden. Anstelle des Prädikates kann ein Integer genommen werden, das dann wie ein Indexwert wirkt.
- Ein Step ist von einer der folgenden Formen:
  - Name-Test; es werden die Knoten mit dem angegebenen Element/Attribut-Name ausgewählt. Attributnamen haben ”@” als Präfix.
  - Kind-Test einer der Formen: processing-instruction(), comment(), text(), node(). Es werden die Knoten des angegebenen Kindes ausgewählt.
  - Axis gefolgt von ”::” gefolgt von Name/Kind-Test.

XPath im Detail ==> Semistrukturiete Daten 1.

Erzeugen von Knoten

- Elemente
  - Elemente können durch Angabe der Tags samt Attribut und Text-Inhalt kreiert werden.
  - Es kann der Element-Konstructor `element { ... }` oder Attribut-Constructor `attribute { ... }` verwendet werden.
  - Element-Name und Attribut-Name, sowie Element-Inhalt und Attribut-Wert können dynamisch mittels eines Anfrage-Ausdrucks berechnet werden.
    - `<aTag> { xQuery Expression } </aTag}`
    - `element aTagName { xQuery Expression }`
    - `element { xQuery Expression } { xQuery Expression }`
- Dokument-Konstruktor: `document { .... }`.

Kombinieren und Restrukturieren von Knoten: FLWOR-Ausdrücke

- `For:` Eine oder mehrere Variablen werden an Ausdrücke gebunden und erzeugen so einen Strom von Tupeln, in dem jedes Tupel jede Variable an einen Wert bindet, der im Resultat des betreffenden Ausdrucks liegt.
- `Let`: Analog zur For-Klausel, jedoch wird jetzt jede Variable an das gesamte Resultat des Ausdrucks gebunden. Alle in einer Let-Klausel eingeführten Variablen definieren so ein einziges Tupel, sofern keine For-Klausel existiert, bzw. erweitern die durch die For-Klausel definierten Tupel.
- `Where`: Die erzeugten Tupel werden gefiltert; nur diejenigen werden weiter betrachtet, die den Ausdruck der Where-Klausel erfüllen.
- `Order by:` Die Tupel werden sortiert.
- `Return:` Die Tupel werden für die Resultatbildung betrachtet.

For- und Let-Klausel

`For` und `Let` kreieren Tupel.

```
for $i in (1,2,3)
return <tuple><i>{ $i }</i></tuple>
```

erzeugt 3 Tupel;

```
let $i := (1,2,3)
return <tuple><i>{ $i }</i></tuple>
```

erzeugt 1 Tupel.

Entsprechend

```
for $i in (1,2,3)
let $j := (1,2,3)
return <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

erzeugt:

```
<tuple><i>1</i><j>1 2 3</j></tuple>
<tuple><i>2</i><j>1 2 3</j></tuple>
<tuple><i>3</i><j>1 2 3</j></tuple>
```

Kartesisches Produkt

```
for $i in (1,2,3)
for $j in  (1,2,3)
return <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

erzeugt:

```
<tuple><i> 1 </i><j> 1 </j></tuple>
<tuple><i> 2 </i><j> 1 </j></tuple>
<tuple><i> 3 </i><j> 1 </j></tuple>
<tuple><i> 1 </i><j> 2 </j></tuple>
<tuple><i> 2 </i><j> 2 </j></tuple>
<tuple><i> 3 </i><j> 2 </j></tuple>
<tuple><i> 1 </i><j> 3 </j></tuple>
<tuple><i> 2 </i><j> 3 </j></tuple>
<tuple><i> 3 </i><j> 3 </j></tuple>
```

Where-Klausel

- Nur die Tupel, die die Where-Klausel passieren, werden für die Return-Klausel betrachtet.
- In der Where-Klausel ist ein beliebiger Ausdruck erlaubt, der zu einem Boole'schen Wahrheitswert ausgerechnet werden kann.

```
for $b in doc("books.xml")//book
let $c := $b//author
where count($c) > 2
return $b/title
```

Order-by-Klausel

Vor Bildung des Resultates (Return-Klausel) werden die Tupel sortiert.

```
for $a in doc("books.xml")//author
order by $a/last descending, $a/first descending
return $a
```

- Nullwerte relationaler Datenbanken `null` können als größter mittels `empty greatest`, bzw. mittels `empty least` als kleinster Wert definiert werden.
- Sind zwei Tupel bzgl. dem Sortierkriterium gleich, kann mittels `stable` die ursprüngliche Reihenfolge im Dokument festgeschrieben werden.
- Wird beispielsweise in einer Let-Klausel eine Sortierung definiert, so kann diese u.U. im Ergebnis verletzt sein, da ’/’ und ’//’-Operatoren die Dokument-Order erzeugen.

```
let $authors := for $a in doc("books.xml")//author
                order by $a/last, $a/first
                return $a
return $authors/last
```

Return-Klausel

- Mittels Element-Konstruktoren kann die Struktur eines Dokumentes geändert werden.

```
for $b in doc("books.xml")//book
return
<quote>{ $b/title, $b/price }</quote>

for $a in doc("books.xml")//author
return
<author>
<Name>{ $a/first, $a/last}</Name>
</author>
```

at-Klausel

- Positions-Variablen können ausgenutzt werden, um innerhalb einer for-Klausel die Position eines Items bzgl. des betreffenden Ausdrucks zu erfassen.

```
for $t at $i in doc("books.xml")//title
return <title pos="{$i}">{string($t)}</title>
```

- Mittels `distinct-values(...)` können Duplikate unter Werten eliminiert werden.

```
let $a := doc("books.xml")//author
for $l in distinct_values($a/last),
    $f in distinct_values($a[last=$l]/first)
return
 <author>
   <last>{$l}</last>
   <first>{$f}</first>
 </author>
```

Verbund

- Mittels `for` und `where` kann ein Verbund berechnet werden.

```
for $t in doc("books.xml")//title,
      $e in doc("reviews.xml")//entry
where $t = $e/title
return
<review>{$t, $e/remarks}</review>


for $t in doc("books.xml")//title
return
<review>
    {$t}
    {
         for $e in doc("reviews.xml")//entry
         where $e/title = $t
         return $e/remarks
    }
</review>
```

Invertieren einer hierarchischen Struktur

```
<listings>
{
for $p in distinct-values(doc("books.xml")//publisher)
order by $p
return
<result>
    {$p}
    {
        for $b in doc("books.xml")/bib/book
        where $b/publisher = $p
        order by $b/title
    }
</result>
}
</listing>
```

Quantoren

- Existenz-Quantor

```
for $b in doc("books.xml")//book
where some $a in $b/author
           satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

- All-Quantor

```
for $b in doc("books.xml")//book
where every $a in $b/author
           satisfies ($a/last="Stevens" and $a/first="W.")
return $b/title
```

Bedingte Ausdrücke

```
for $b in doc("books.xml")//book
return
<book>
{ $b/title }
{
    for $a at $i in $b/author
    where $i <= 2
    return <author>{string($a/last), ", ",
                    string($a/first)}
           </author>
}
{
    if (count($b/author) > 2)
    then <author>et al.</author>
    else {}
}
</book>
```

Operatoren

Bereitstellen von Werten für die Anwendung von Operatoren in arithmetischen und Vergleichsausdrücken.

- einzelne Items: Typed value extraction.

  Element-Namen in Ausdrücken werden durch ihren getypten Wert ersetzt.

  ```
  doc("book.xml")/bib/book/author[last="Stevens"]
  ```

- Sequenzen von Items: Atomization liefert eine entsprechende Sequenz von getypten Werten.

  ```
  avg(1, <e>2</e>, <e>3</e>)
  ```

- Arithmetische Operatoren
  - Analog zu SQL können Nullwerte resultieren: `2 + ()`
  - Ungetypte Operanden werden mittels eines impliziten `cast` nach `double` gewandelt.

    ```
    let $p := doc("books.xml")//price
    return $p[1] + $p[2]
    ```

- Vergleichsoperatoren: value comparison, general comparison, node comparison, order comparison.
  - value comparison (eq, ne, lt, le, gt, ge): Vergleich zweier atomarer Werte.

```
 for $b in
doc("books.xml")//book where $b/title eq "Data on the Web" return
$b/price
```

```
for $b in doc("books.xml")//book
where $b/author/last eq "Stevens"
return $b/title
```

    Vergleich zweier Strings; anderenfalls explizites `cast` erforderlich.

```
for $b in doc("books.xml")//book
where decimal($b/price) eq 100.00
return $b/title
```

- general comparison (=, !=, <, <=, >, >=): Vergleich zweier Sequenzen atomarer Werte.

  ”Any value on the left matches any value on the right.”

```
for $b in doc("books.xml")//book
where $b/author/last = "Stevens"
return $b/title
```

  Implizites `cast` wird gegebenenfalls versucht.

```
for $b in doc("books.xml")//book
where $b/price = 100.00
return $b/title
```

  Achtung! ” = ” ist nicht transitiv.

```
for $b in doc("books.xml")//book
where $b/author/last = "Suciu"
      and $b/author/first = "Serge"
return $b


for $b in doc("books.xml")//book
for $a in $b/author
where $a/last = "Suciu"
      and $a/first = "Serge"
return $b
```

- node comparison (`is`): Vergleich zweier Knoten.

```
Let $b1 :=for $b  in doc("books.xml")//book
       order by count($b/author) + count($b/editor)
       return $b

Let $b2 :=for $b  in doc("books.xml")//book
        order by count($b/price)
        return $b
return $b1[last()] is $b2[last()]
```

- order comparison («, »): Vergleich zweier Knoten bzgl. der Document Order.

```
for $b  in doc("books.xml")//book
Let $a := ($b/author)[1],
      $sa := ($b/author)[last = "Abiteboul"]
where $a << $aa
return $b
```

Sequenz Operatoren: `union, intersect, except`

Angewendet auf zwei Sequenzen wird eine Resultat-Sequenz ohne Duplikat-Knoten in Dokument-Order erzeugt.

- Union mittels |

```
 let $l := distinct-values(doc("books.xml")//
                           (author | editor)/last)
order by $l
return <last>  {$l} </last>
```

- Union mittels `union`

```
for $b in doc("books.xml")//book
let $a := ($b/author union $b/editor)[1]
order by $a/last, $a/first
return $b
```

except, intersect

```
for $b in doc("books.xml")//book
where $b/title = "TCO/IP Illustrated"
return
    <book>
    {$b/@*}
    {$b/* except $b/price}
    </book>
```

XQuery (fortgesetzt)

Built-in Funktionen; einige Beispiele:

- ```
  let $b := doc("books.xml")//book
  let avg := average($b//price)
  return $b[price > $avg]
  ```
- ```
  for $b in doc("books.xml")//book
  where not(some $a in $b/author satisfies $a/last="stevens")
  return $b
  ```
- ```
  for $b in doc("books.xml")//book
  where not(empty($b/author))
  return $b
  ```
- ```
  for $b in doc("books.xml")//book
  where exists($b/author)
  return $b
  ```

User-Defi ned Functions; ein Beispiel.

```
define function toc($param as element()) as element()* {
 for $section in $param/section
 return
 <section>
    {$section/@*, $section/title, toc($section)}
 </section>

<toc> {
    for $s in doc("xquery-book.xml")/book
    return toc($s)
}
</toc>
```

XQuery und Typen

- built-in Typen,
- von einem Schema importierte Typen.

Basic XQuery, mindestens vorausgesetzt für jede XQuery Implementierung, verlangt

- XML Schema Definitionen können importiert werden,
- statische Typprüfung bzgl. dem importierten Schema ist möglich.

Beispiel: keine Typangaben erforderlich

```
define function reverse ($items)
 {
 let $count := count($items)
 for $i in 0 to $count
 return $items[$count - $i]
 }

reverse(1 to 5)
```

Beispiel: keine spezielle Typangabe erforderlich

```
define function is-document-element ($e as element())
 as xs:boolean
 {
 if ($e/.. instance of document-node())
   then true()
   else false()
 }
```

Beispiel: built-in types/Typkonversion

```
avg(doc("books.xml")/bib/book/price)
```

Beipiel Schema-Import erforderlich

```
define function books-by-author($author)
 {
 for $b in doc("books.xml")/bib/book
 where some $ba in $b/author satisfies
   ($ba/last = $author/last and $ba/first = $author/first)
 order by $b/title
 return $b/title
 }
```

Aufruf `books-by-author(/bib/book)` möglich.

Typprüfung verlangt Import eines Schema.

```
import schema "urn:examples:xmp:bib" at
              "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

define function books-by-author($a as element(author))
 as element(title)*
 {
 for $b in doc("books.xml")/bib/book
 where some $ba in $b/author satisfies
   ($ba/last = $a/last and $ba/first = $a/first)
 order by $b/title
 return $b/title
 }
```

Aufruf mit `author`-Element erzwungen.

Beispiel statische Prüfung.

```
import schema "urn:examples:xmp:bib" at
              "c:/dev/schemas/eg/bib.xsd"
default element namespace = "urn:examples:xmp:bib"

define function books-by-author($a as element($author))
 as element(title)*
 {
 for $b in doc("books.xml")/bib/element(book)
 where some $ba in $b/author satisfies
   ($ba/last = $a/last and $ba/first = $a/first)
 order by $b/title
 return $b/title
 }
```

$b enthält garantiert book-Element gemäß dem importierten Schema.

Tippfehler

```
    ($ba/last = $a/last and $ba/fist = $a/first)
```

führt zu statisch erkanntem Typfehler!

Werte in XQuery sind Sequenzen; ihr Typ ist somit ein *Sequenz-Typ*.

- Mittels sogenannter *occurence indicators* kann die Häufigkeit des Auftretens in einer Sequenz festgelegt werden:
  - `+`: ein- oder mehrmals; `element()+`,
  - `*`: kein- oder mehrmals; `node()*`,
  - `?`: ein- oder keinmal, `xs:integer?`.

XQuery-Ausdrücke können Elemente/Attribute auf Typen testen; ein erfolgreicher Test wird als *Match* bezeichnet.

- `element(aName, aType)`, bzw. `attribute(aName, aType)`,
- `element(aContextPath, aType)`, bzw. `attribute(aContextPath, aType)`.
- `element(aName)`, bzw. `attribute(aName)`, bzw. `element(aContextPath)`, bzw. `attribute(aContextPath)`. In diesem Fall wird der Typ gemäß dem betreffenden Schema genommen.
- `element(n, person nillable)` bewirkt u.a. einen Match mit `< n xsi:nil='true'/ >`.

Sequence Type Declaration	What it Matches
`element(creator, person)`	An element named `creator` of type `person`
`element(creator)`	Any element of type named `creator` of type `xs:string` – the type declared for `creator` in the schema
`element(*, person)`	Any element of type `person`
`element(book/price)`	An element named price of type `currency` – the type declared for `price` elements inside a book element
`element(type(person)/last)`	An element named last of type `xs:string` – the type declared for `last` elements inside the `person` type
`attribute(@price, currency)`	An attribute named `price` of type `currency`
`attribute(book/@isbn)`	An attribute named `isbn` of type `isbn` – the type declared for `isbn` attributes in a `book` element
`attribute(@*, currency)`	Any attribute of type `currency`
`currency`	A value of the user-defined type `currency`

Arbeiten mit Typen

```
define function discount-price($b as element(book))
  as xs:decimal
  {
  0.80 * $b//price
  }

 for $b in doc("books.xml")//book
 where $b/title = "Data on the Web"
 return
   <result>
     {
       $b/title, <price>{ discount-price($b) }</price>
     }
   </result>
```

gefährlich, aber korrekt!

besser:

```
define function discount-price($b as xs:decimal)
  as xs:decimal
  {
  0.80 * $b
  }

 for $b in doc("books.xml")//book
 where $b/title = "Data on the Web"
 return
   <result>
     {
       $b/title, <price>{ discount-price($b/price) }</price>
     }
   </result>

  let $w := <foo> 12.34 </foo>
  return discount-price($w)
```

weitere Sprachelement zu Typen

- `instance of`
- `typeswitch`
- `treat as`
- `validate skip`