# XML and Databases

# Chapter 14: XQuery II

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Winter 2022/23

http://www.informatik.uni-halle.de/~brass/xml22/

# Objectives

After completing this chapter, you should be able to:

- read and understand queries in XQuery.
- write queries to XML data in XQuery.
- enumerate and explain the clauses of FLWOR expressions.
- explain the use of direct and computed constructors.
- compare XQuery with SQL.

# Contents

# Example Document (1)

**STUDENTS**

| SID | FIRST | LAST | EMAIL |
|---|---|---|---|
| 101 | Ann | Smith | ... |
| 102 | David | Jones | NULL |
| 103 | Paul | Miller | ... |
| 104 | Maria | Brown | ... |

**RESULTS**

| SID | CAT | ENO | POINTS |
|---|---|---|---|
| 101 | H | 1 | 10 |
| 101 | H | 2 | 8 |
| 101 | M | 1 | 12 |
| 102 | H | 1 | 9 |
| 102 | H | 2 | 9 |
| 102 | M | 1 | 10 |
| 103 | H | 1 | 5 |
| 103 | M | 1 | 7 |

**EXERCISES**

| CAT | ENO | TOPIC | MAXPT |
|---|---|---|---|
| H | 1 | ER | 10 |
| H | 2 | SQL | 10 |
| M | 1 | SQL | 14 |

# Example Document (2)

- Translation to XML with data values in elements:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>101</SID>
      <FIRST>Ann</FIRST>
      <LAST>Smith</LAST>
    </STUDENT>
    ...
  </STUDENTS>
  ...
</GRADES-DB>
```

# Simple Query (1)

- Print all homework results of student 101:

```
SELECT ENO, POINTS
FROM   RESULTS
WHERE  CAT = 'H' AND SID = 101
```

| ENO | POINTS |
|---|---|
| 1 | 10 |
| 2 | 8 |

- Solution 1 in XQuery:

```
<STUD_101>{
    for $r in /GRADES-DB/RESULTS/RESULT
    where $r/CAT = 'H' and $r/SID = 101
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

Whereas the result table structure in SQL is fixed, there are many ways to structure the output in XML. The query has to specify this.

# Simple Query (2)

- Result:

```
<STUD_101>
    <HW>
        <ENO>1</ENO>
        <POINTS>10</POINTS>
    </HW>
    <HW>
        <ENO>2</ENO>
        <POINTS>8</POINTS>
    </HW>
</STUD_101>
```

Of course, the line breaks and indentation depend on the output serialization. This is the result of BaseX. AltovaXML by default writes everything in one line, but "`/oi yes`" ("outputindent") gives the above.

# Simple Query (3)

- Solution 2 (predicate under `for` instead of `where`-clause, requires fewer variable references):

```
<STUD_101>{
    for $r in
        //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

- Solution 3 (with context instead of variable):

```
<STUD_101>{
    //RESULT[CAT = 'H' and SID = 101]/
        <HW>{ENO, POINTS}</HW>
}</STUD_101>
```

# Simple Query (4)

- The last example shows that the use of constructors is not limited to the `return`-clause of FLWOR-expressions.
- In the grammar, constructors are "Primary Expressions", i.e. on the same level as datatype literals or variables.

  [https://www.w3.org/TR/xquery/#nt-bnf]

  As in XPath, a "StepExpr" in a path expression can not only be an "AxisStep", but also a "FilterExpr", which is a "PrimaryExpr" optionally followed by predicates. This again shows why the grammar rules of XPath had to be repeated in the XQuery grammar: Constructors in primary expressions are new in XQuery, but this has consequences for standard path expressions.

# Problem with Namespaces (1)

- Input document with link to XML Schema:

```
<GRADES-DB xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="grades.xsd">
```

- The namespace definition for `xsi` implicitly applies to all element nodes of the input document.
- Since the `ENO` and `POINTS`-nodes are copied from the input document, each such node shows this namespace explicitly in the output, e.g.

```
<ENO xmlns:xsi="...">1<ENO>
<POINTS xmlns:xsi="...">10<POINTS>
```

# Problem with Namespaces (2)

- It should be possible to avoid this as follows:

```
declare copy-namespaces no-preserve, no-inherit;
<STUD_101>{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

`inherit` or `no-inherit` is actually not important for this task, but the syntax forces one to specify it together with `no-preserve`. The standard says: "If `copy-namespaces` mode specifies `no-preserve`, the new copy retains only those in-scope namespaces of the original element that are used in the names of the element and its attributes."

- But neither BaseX nor AltovaXML remove the namespace declarations.

# Problem with Namespaces (3)

- If one specifies this namespace in the outer element of the query, the output serialization does not repeat it in each copied element:

```
<STUD_101 xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance">{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return <HW>{$r/ENO, $r/POINTS}</HW>
}</STUD_101>
```

Now the namespace is declared for all nodes in the output document, so when the nodes from the input document are copied, they are in scope of the namespace declaration, and it is not necessary to explicitly repeat the inherited declaration.

# Problem with Namespaces (4)

- A solution is to construct new element nodes with the same name and the same contents:

```
<STUD_101>{
    for $r in //RESULT[CAT = 'H' and SID = 101]
    return
        <HW>
            <ENO>{data($r/ENO)}</ENO>
            <POINTS>{data($r/POINTS)}</POINTS>
        </HW>
}</STUD_101>
```

With the same query structure, one could also generate an HTML table: Replace `STUD_101` bei `table`, `HW` by `tr`, and the new `ENO` and `POINTS` tags by `td`. One could even add a headline.

# Joins (1)

- "Print the names of all students who have at least 9 points in both, Homework 1 and Homework 2":

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS H1, RESULTS H2
WHERE  S.SID = H1.SID AND S.SID = H2.SID
AND    H1.CAT = 'H' AND H1.ENO = 1
AND    H2.CAT = 'H' AND H2.ENO = 2
AND    H1.POINTS >= 9 AND H2.POINTS >= 9
```

| FIRST | LAST |
|---|---|
| David | Jones |

# Joins (2)

- XQuery:

```
<ANSWER>{
for   $s  in //STUDENT,
      $h1 in //RESULT, $h2 in //RESULT
where $s/SID = $h1/SID and $s/SID = $h2/SID
and   $h1/CAT = 'H' and $h1/ENO = 1
and   $h2/CAT = 'H' and $h2/ENO = 2
and   $h1/POINTS >= 9 and $h2/POINTS >= 9
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

```
Query Result: <ANSWER>
                  <ROW FIRST="David" LAST="Jones"/>
              </ANSWER>
```

# Joins (3)

- One can move parts of the condition to predicates in the `for`-clause:

```
<ANSWER>{
for   $s  in //STUDENT,
      $h1 in //RESULT[CAT = 'H' and ENO = 1],
      $h2 in //RESULT[CAT = 'H' and ENO = 2]
where $s/SID = $h1/SID and $s/SID = $h2/SID
and   $h1/POINTS >= 9 and $h2/POINTS >= 9
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

# Joins (4)

- One can put the entire `where`-condition into predicates in the `for`-clause (similar to joins under `FROM`):

```
<ANSWER>{
for $s  in //STUDENT,
    $h1 in //RESULT[CAT = 'H'] [ENO = 1]
                   [SID = $s/SID]
                   [POINTS >= 9],
    $h2 in //RESULT[CAT = 'H' and ENO = 2 and
                    SID = $s/SID and
                    POINTS >= 9]
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

# Joins (5)

- An XML document that directly corresponds to the table structure of a relational database does not make optimal use of XML's tree structure.
- If the RESULT elements of a student were nested inside the STUDENT element, one would not need an explicit join on SID:

```
for $s  in //STUDENT,
    $h1 in $s/RESULT[CAT = 'H'] [ENO = 1]
                    [number(POINTS) >= 9],
    $h2 in $s/RESULT[CAT = 'H'] [ENO = 2]
                    [number(POINTS) >= 9]
...
```

# Numeric Comparisons (1)

- Who has at least as many points for Homework 1 as Student 101?

```
<ANSWER>{
for   $s in //STUDENT,
      $x in //RESULT[CAT='H' and ENO=1],
      $y in //RESULT[CAT='H' and ENO=1]
where $x/SID = $s/SID and $s/SID != 101
and   $y/SID = 101
and   number($x/POINTS) >= number($y/POINTS)
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

# Numeric Comparisons (2)

- If the document is not validated, attribute values and values of elements with simple content are of type `untypedAtomic`.
- When the comparison is done with a value of numeric type (e.g. a constant as in earlier examples), a numeric comparison is done.

  The `untypedAtomic` value is converted to the more specific type.
- If two `untypedAtomic` values are compared, a string comparison is done.

  To get a numeric comparison, one must use `number(...)` on at least one side of the comparison.

# NOT EXISTS (1)

- "Print the names of all students who have not yet submitted a homework":

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S
WHERE  NOT EXISTS(SELECT *
                  FROM   RESULTS R
                  WHERE  R.SID = S.SID
                  AND    R.CAT = 'H')
```

| FIRST | LAST |
|---|---|
| Maria | Brown |

# NOT EXISTS (2)

- Note that `not(...)` computes the effective boolean value of its argument, and thus can be used to check for existence of a node:

```
<ANSWER>{
for $s in //STUDENT
where not(//RESULT[SID=$s/SID and CAT='H'])
return <ROW FIRST = "{$s/FIRST}"
           LAST  = "{$s/LAST}" />
}</ANSWER>
```

```
Query Result: <ANSWER>
                  <ROW FIRST="Maria" LAST="Brown"/>
              </ANSWER>
```

# NOT EXISTS (3)

- Instead of `not(...)` one can also use

```
count(//RESULT[SID=$s/SID][CAT='H']) = 0
```

For sequences of nodes, this is equivalent (not for atomic values).

- One can also use an explicit quantifier:

```
<ANSWER>{
for $s in //STUDENT
where every $r in //RESULT[SID = $s/SID]
                satisfies CAT != 'H'
return <ROW FIRST = "{$s/FIRST}"
            LAST  = "{$s/LAST}" />
}</ANSWER>
```

# Universal Quantification (1)

- "Print the names of all students who have solved all homeworks in the database":

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT *
       FROM  EXERCISES E
       WHERE E.CAT = 'H'
       AND   NOT EXISTS (SELECT *
             FROM  RESULTS R
             WHERE R.SID = S.SID
             AND   R.CAT = 'H'
             AND   R.ENO = E.ENO))
```

# Universal Quantification (2)

- In XQuery, the "for all" can be directly expressed:

```
<STUDENTS_WITH_ALL_HOMEWORKS>{
  for $s in //STUDENT
  where
    every $e in //EXERCISE[CAT='H'] satisfies
    //RESULT[SID=$s/SID][CAT='H'][ENO=$e/ENO]
  return <ROW FIRST = "{$s/FIRST}"
              LAST  = "{$s/LAST}" />
}</STUDENTS_WITH_ALL_HOMEWORKS>
```

```
Query Result: <STUDENTS_WITH_ALL_HOMEWORKSANSWER>
                  <ROW FIRST="Ann" LAST="Smith"/>
                  <ROW FIRST="David" LAST="Jones"/>
                </STUDENTS_WITH_ALL_HOMEWORKSANSWER>
```

# LIKE (1)

- "Print the names of all students who have an email address from `acm.org`":

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  EMAIL LIKE '%@acm.org'
```

| FIRST | LAST |
|---|---|
| Ann | Smith |

- The function library contains
  - `contains(`$s_1$`,` $s_2$`)`: Substring test.
  - `starts-with(`$s_1$`,` $s_2$`)`: Prefix test.
  - `ends-with(`$s_1$`,` $s_2$`)`: Suffix test (this example).
  - `matches(`$s$`,` $p$`)`: Regular expression test.

See [https://www.w3.org/TR/xpath-functions/].

# LIKE (2)

- Solution with `ends-with`:

```
for $s in //STUDENT[ends-with(EMAIL, '@acm.org')]
return ...
```

- Solution with `matches`:

```
for $s in //STUDENT
          [matches(EMAIL, '^.*@acm\.org$')]
return ...
```

$ matches the end of the string (or a line end in multi-line mode). It is necessary, because otherwise the regular expression could match any substring.
In the same way, ^ ensures that the match must begin at the start of the string.
Of course, `^.*` could be left out, but in this way, the example demonstrates a match of the entire string. The meta character "." must be escaped with "\".
Case-insensitive matching is done with a third argument `'i'` ("flags").

# Duplicate Elimination (1)

- "Print the numbers of all homeworks for which there is at least one graded submission (result)":

```
SELECT DISTINCT ENO
FROM   RESULTS
WHERE  CAT = 'H'
```

| ENO |
|---|
| 1 |
| 2 |

- Duplicates in a sequence of atomic values can be removed with the function `distinct-values`.
- Often, existential quantification helps.

# Duplicate Elimination (2)

- Solution with `distinct-values`:

```
<RESULT>{
let $s := //RESULT[CAT = 'H']/ENO
for $n in distinct-values($s)
return <HOMEWORK>{$n}</HOMEWORK>
}</RESULT>
```

Note that `distinct-values` applies atomization to its argument.

- Solution with existential quantification:

```
for $h in //EXERCISE[CAT = 'H']
where exists(//RESULT[CAT = 'H'][ENO = $h/ENO])
return ...
```

# Simple Aggregations (1)

- "How many students are in the database?":

```
SELECT COUNT(*)
FROM   STUDENTS
```

| COUNT(*) |
|---|
| 4 |

- The library offers the usual aggregation functions:
  - `count(s)`: Number of items in a sequence
  - `sum(s[,z])`: Sum (with result $z$ for empty seq.)
  - `avg(s)`: Average.
  - `min(s)`: Minimum.
  - `max(s)`: Maximum.

# Simple Aggregations (2)

- This is simple in XQuery, too:

```
<NUM_STUDENTS>
  {count(//STUDENT)}
</NUM_STUDENTS>
```

- Note that in XQuery, aggregation functions can be used under `where`, which would be forbidden in SQL.

  The reason is that in XQuery, the argument of the aggregation function computes the set (sequence) of values to be aggregated. In SQL, the argument is only an attribute, and the aggregation is over variable assignments generated by the FROM-clause.

# Simple Aggregations (3)

- How many distinct topics are there?

```
SELECT COUNT(DISTINCT TOPIC)
FROM   EXERCISES
```

| COUNT(...) |
|---|
| 2 |

- In XQuery:

```
count(distinct-values(//EXERCISE/TOPIC))
```

- In SQL, null values are not counted:

```
SELECT COUNT(EMAIL)
FROM   STUDENTS
```

| COUNT(EMAIL) |
|---|
| 3 |

- In XQuery, this happend automatically.

The path `//STUDENT/EMAIL` selects only existent email elements.

# GROUP BY (1)

- "Print for every student the total number of homework points."

```
SELECT S.FIRST, S.LAST, SUM(R.POINTS)
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID AND R.CAT = 'H'
GROUP  BY S.SID, S.FIRST, S.LAST
```

| FIRST | LAST | SUM(POINTS) |
|---|---|---|
| Ann | Smith | 18 |
| David | Jones | 18 |
| Paul | Miller | 5 |

# GROUP BY (2)

- If we want the same in XQuery, we must explicitly exclude students without homeworks:

```
<STUDENTS_WITH_SUM_HW_POINTS>
{
  for $s in //STUDENT
  let $p := //RESULT[SID=$s/SID][CAT='H']/POINTS
  where exists($p)
  return <ROW FIRST="{$s/FIRST}" LAST="{$s/LAST}"
              SUM="{sum($p)}" />
}
</STUDENTS_WITH_SUM_HW_POINTS>
```

Producing an output for a student without submitted homeworks can be a bug or a feature. If it is required, the SQL query becomes longer, and the XQuery query becomes shorter.

# GROUP BY (3)

- SQL query with 0 points for students who have not yet submitted any homework:

```
SELECT   S.FIRST, S.LAST,
         COALESCE(SUM(R.POINTS), 0)
FROM     STUDENTS S LEFT JOIN RESULTS R
         ON S.SID = R.SID AND R.CAT = 'H'
GROUP BY S.SID, S.FIRST, S.LAST
```

- Note that the XPath function `sum` produces a numeric `0` for the empty sequence, whereas SQL produces a null value in this case.

  Here the input to the SQL function is not empty, but consists of a null value. The problem appears in simple aggregations.

# Restructuring the Data (1)

- Suppose we want to remove the elements for the relations (like STUDENTS), and put the tuple elements directly below GRADES-DB:

```
<GRADES-DB>{
    for $e in /GRADES-DB/*/*
    return $e
}</GRADES-DB>
```

- This gives

```
<GRADES-DB>
    <STUDENT>
        <SID>101</SID>
        ...
```

# Restructuring the Data (2)

- The opposite transformation (grouping tuple elements by relation) is also possible:

```
<GRADES-DB>
    <STUDENTS>{
        for $s in /GRADES-DB/STUDENT
        return $s
    }</STUDENTS>
    <EXERCISES>{
        for $e in /GRADES-DB/EXERCISE
        return $e
    }</EXERCISES>
    ...
</GRADES-DB>
```

# Restructuring the Data (3)

- Nesting results under students (data in attributes):

```
<GRADES-DB>{
    for $s in //STUDENT
    return element STUDENT {
        for $d in $s/*
        return attribute {name($d)} {data($d)},
        for $r in //RESULT[SID=$s/SID]
        return element RESULT {
            for $a in $r/*
            where name($a) ne "SID"
            return attribute {name($a)} {data($a)}
        }
    },
    ...(: Copy/transform EXERCISE data :)
}</GRADES-DB>
```

# Restructuring the Data (4)

- The output looks as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<GRADES-DB>
  <STUDENT SID='101' FIRST='Ann' LAST='Smith'>
    <RESULT CAT='H' ENO='1' POINTS='10'/>
    <RESULT CAT='H' ENO='2' POINTS='8'/>
    <RESULT CAT='M' ENO='1' POINTS='12'/>
  </STUDENT>
  <STUDENT SID='102' FIRST='David' LAST='Jones'>
    ...
  </STUDENT>
  ...
</GRADES-DB>
```

# Contents

# Overall Syntax (1)

- The basic XQuery unit is a module.
- A module can be
  - a library module (contains mainly function declarations),
  - a main module (contains mainly the query).
- Each module may optionally start with a version declaration:

```
xquery version "1.0";
```

One can also specify the encoding, but the treatment of this is implementation-dependent: `xquery version "1.0" encoding "utf-8";`

# Overall Syntax (2)

- A main module consists of a prolog (which can be empty) and the query ("QueryBody").
- A library module consists of a module declaration and a prolog.
- The prolog can contain
  - First an arbitrary sequence of namespace declarations, module import commands (for schemas and modules), and XQuery parameter settings,
  - and then an arbitrary sequence of variable, function, and option declarations.

# Overall Syntax (3)

- The query itself ("QueryBody") is an expression.
- XPath-expressions are also XQuery-expressions.

  However, the grammar in the XQuery standard completely defines expressions. Basically, XPath is a restricted version of XQuery. Since XQuery has extensions in many places, it was not possible to simply embedd an XPath expression as defined in the XPath standard.

- As in XPath, all data values are sequences of items, where items are atomic values or nodes.
- Expressions can be arbitrarily nested.

  While only recent SQL DBMS support the use of an SQL query with one result value as a term, the arbitrary nesting was a basic design principle in XQuery. It is sometimes called a funtional language.

# Expressions (1)

- On the top level, an expression consists of one or more subexpressions ("ExprSingle") separated by "`,`" (sequence concatenation operator).
- On the next level, an expression ("ExprSingle") is
  - a FLWOR-expression,
  - a quantified expression (`some`, `every`)
  - a typeswitch expression (see below)
  - an `if`-expression,
  - or an expression with the usual logical, comparison and arithmetic operators (see below).

# Expressions (2)

- In comparison, the XPath 2.0 grammar has
  - a `for`-expression instead of the `FLWOR`-expression,
  - no `typeswitch` expression.
- Note that the `for`-expressions in XPath 2.0 are valid FLWOR-expressions in XQuery:
  - They have only the `for` and the `return` part.

    It is legal in XQuery to leave out the other parts.
  - The `for`-clause is simplified: XQuery permits to declare a type for the variable, and to add a positional variable (see below).

# Expressions (3)

- The grammars for XQuery and XPath 2.0 are very similar (they are generated from a single source, only some possibilities are missing in XPath or replaced by simpler mechanisms).
- Continuing the comparison, one finds that
  - Quantified expressions (`some`, `every`) permit a type declaration for the variable in XQuery.

    In XPath, no such type declaration is possible. In XQuery, it is optional (thus, XPath is still a subset of XQuery).

# Expressions (4)

- The "valueExpression" (Argument of unary + and −, i.e. at the end of the operator hierarchy) is a path expression in XPath. In XQuery, there are two additional possibilities:
  - `validate (strict|lax) { ⟨Expression⟩ }`

    The expression must evaluate to exactly one document or element node. It is treated as an XML infoset (i.e. existing type annotations are ignored), validated according to the "in-scope schema definitions", and a new tree is built from the PSVI. However, the "Schema Import Feature" is optional in XQuery.
  - An "extension expression" with a pragma: `(# ...#) { ⟨Expression⟩ }`

# Expressions (5)

- As explained above, the XPath grammar permits the `namespace` axis, with is not supported in XQuery.

  But because it can be supported only in an inefficient way, it is anyway no longer recommended to use it.

- The next difference is in the "Primary Expression":
  - Both languages permit numeric and string literals, variable references, expressions in `(...)`, the context item "`.`", and function calls.
  - XQuery permits in addition constructors (see below), and "`ordered|unordered { ⟨Expression⟩ }`".

# Operator Precedences (1)

| Prio | Operator | Assoc. |
|---|---|---|
| 1 | , (comma) | left |
| 2 | := (assignment) | right |
| 3 | for, some, every, typeswitch, if | left |
| 4 | or | left |
| 5 | and | left |
| 6 | eq,ne,lt,le,gt,ge,=,!=,<,<=,>,>=,is,<<,>> | left |
| 7 | to | left |
| 8 | +, - | left |
| 9 | *, div, idiv, mod | left |
| 10 | union, \| | left |

(continued on next slide)

# Operator Precedences (2)

(continued from previous slide)

| Prio | Operator | Assoc. |
|---|---|---|
| 11 | `intersect`, `except` | left |
| 12 | `instance of` | left |
| 13 | `treat as` | left |
| 14 | `castable` | left |
| 15 | `cast` | left |
| 16 | - (unary), + (unary) | right |
| 17 | ?, *, + (Occurrence Indicators) | left |
| 18 | /, // | left |
| 19 | `[ ]`, `( )`, {} | left |

Only differences (additions) to XPath: `:=`, `typeswitch`.

# typeswitch-Expression (1)

- The typeswitch-expression permits to check the dynamic type of an expression, and to distinguish different cases based on this type:

```
typeswitch($cust/address)
  case $a as element(*,USAddr)  return $a/state
  case $a as element(*,CanAddr) return $a/province
  case    element(*,GermanAddr) return ()
  default return fn:error("Unknown address type")
```

- element(*,USAddr) matches any non-nilled element node with type annotation USAddr.

  Or a type derived from that. This example needs schema validation.

# typeswitch-Expression (2)

- The first `case`-clause with a matching type is selected, or the default clause if non matches.
- A variable must be declared in the `case` only if the value of the original expression is needed to compute the `return` value.

  The scope of this variable declaration is this single case. Different cases can declare variables with the same name.
- The same effect can be achieved with conditional expressions (`if`) and "`instance of`".

  "`treat as`" is necessary in addition to use the value as a value of its real type. So in the end, the `typeswitch` simplifies the expression.

# Contents

# Namespaces

- Namespaces can be defined in the Prolog:
  - `declare namespace Prefix = "URI";`
  - `declare default element namespace "URI";`
  - `declare default function namespace "URI";`
- The following namespace prefixes are predeclared:
  - `xml = http://www.w3.org/XML/1998/namespace`
  - `xs = http://www.w3.org/2001/XMLSchema`
  - `xsi = http://www.w3.org/2001/XMLSchema-instance`
  - `fn = http://www.w3.org/2005/xpath-functions`
  - `local = http://www.w3.org/2005/xquery-local-functions`

# User-Defined Functions (1)

- One can declare functions in the prolog of the main module (i.e. before the query) and library modules.
- Functions must be in a namespace, but for functions declared in the main module XQuery defines the namespace prefix `local`.
- A simple example is:

```
declare function local:inc($n as xs:integer)
        as xs:integer
        { $n+1 };
local:inc(1) (: This is the query :)
```

# User-Defined Functions (2)

- Thus, a function declaration consists of:
  - The keywords "`declare function`",
  - the name of the function with namespace prefix,
  - a comma-separated list of parameter declarations in `(...)`, each consisting of a variable and optionally the keyword "`as`" and a sequence-type,
  - optionally, a specification of the return type: the keyword "`as`" and a sequence-type,
  - and body of the function: an expression in `{...}`
  - and finally a "`;`".

# User-Defined Functions (3)

- If the types are not specified, `item()*` is assumed (the most general type).
- Instead of a function body, one can also specify the keyword "`external`".

  It is implementation-dependent if an how functions written in some other language (e.g., C) can be linked to the XQuery evaluator.
- Functions can be recursive.

  And also mutually recursive. XQuery becomes in this way computationally complete, but then it cannot guarantee termination.

# References


- Scott Boag, Don Chamberlin, Mary F. Fernńndez, Daniela Florescu, Jonathan Robie, Jérôme Siméon (Eds.):
  XQuery 1.0: An XML Query Language.
  W3C Recommendation 23 January 2007.
  [http://www.w3.org/TR/xquery/]
- Wolfgang Lehner, Harald Schöning:
  XQuery. Grundlagen und fortgeschrittene Methoden.
  dpunkt.verlag, 2004, ISBN 3-89864-266-6, 290 Seiten.
  [http://www.xquery-buch.de/]
- Howard Katz (Ed.), Don Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jérôme Siméon, Jim Tivy, Philip Wadler:
  XQuery from the Experts. A Guide to the W3C XML Query Language.
  Pearson Education Inc., 2004, ISBN 0-321-18060-7, 484 pages.
- Jim Melton, Stephen Buxton:
  Querying XML: XQuery, XPath, and SQL/XML in Context.
  Morgan Kaufmann/Elsevier, 2006, ISBN 1-55860-711-0, 815 pages.
- Rudolf Jansen:
  XQuery: Eine praxisorientierte Einführung.
  Software & Support Verlag GmbH, 2004, ISBN 3-935042-65-5, 167 Seiten.