XML and Databases

Chapter 12: XPath III: Functions

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:

- write XPath expressions for a given application.
- explain what is the result of a given XPath expression with respect to a given XML data file.
- explain how comparisons are done, and why XPath has two sets of comparison operators (e.g. `=` vs. `eq`).
- define “atomization”, “effective boolean value”.
- enumerate some axes and explain abbreviations.
- explain features needed for static type checking.

Contents

General Remarks (1)

- Many functions permit the empty sequence as input. E.g. argument type "`node()?`" means a sequence consisting of 0 or 1 nodes.
  - Most functions return the empty sequence if the input is the empty sequence.

    E.g. `node-name` has result type `QName?`, which means that the result is a `QName` or the empty sequence. The empty sequence is returned if the input is the empty sequence, but also for nodes that have no name, i.e. text nodes, document nodes, or comment nodes.
  - Some functions return the empty string if the input is the empty sequence.

    An example of this is `name`. Its return type is `xs:string`, therefore it is clear that it cannot return the empty sequence.

General Remarks (2)

- There can be several functions with the same name, but different number of arguments (overloading).

  Functions that differ only in argument types were avoided if possible. However, they are sometimes needed for numeric functions, and also seldom for backward compatibility.

- A typical case is a function with an optional argument, e.g.
  - `name(n)`: Returns the name of node `n`.
  - `name()`: Returns the name of the context node.

    If the context item is no node, this gives an error.

General Remarks (3)

- Type promotion:
  - If a function is declared with an argument of type `double`, one can call it with an argument of type `decimal` (or any of its subtypes, e.g. `integer`).

    The argument value is automatically converted to a `double` (possibly with a loss of precision).
  - In the same way, a `decimal` value is automatically converted to `float` value if necessary.
  - Also `float` can be converted to `double`.
  - `anyURI` is converted to `string` if needed.

General Remarks (4)

- Type substitution:
  - An element of subtype can be used wherever an element of the supertype is required.
  - E.g., if a function is declared with an argument of type `decimal`, one can pass an `integer` value (`integer` is a subtype of `decimal`).
  - This is not type promotion, because the value is not changed/converted: It remains an `integer`.

    E.g. if the parameter `$n` is declared as `decimal`, but the actual value is an `integer`, "`$p instance of xs:integer`" inside the function returns true.

General Remarks (5)

- More Function Conversion Rules:
  - If the declared argument type is a sequence of atomic values, atomization is applied, i.e. the typed value of nodes is taken.

    E.g. if an attribute is declared of type `integer`, one can specify the attribute node as argument to a function that requires an `integer`: The node is automatically converted to its value.
  - If an atomic value is of type `xs:untypedAtomic` (resulting from a non-validated XML document), it is converted to the required type.

    If a function has variants for different numeric types, `double` is chosen.

General Remarks (6)

- Additional Conversions in XPath 1.0 Compatibility Mode:
  - A sequence can be automatically converted to its first element.

    For XPath 2.0, it is an error to pass a sequence with more than one element if the function accepts only a single value.
  - For the expected types `string` or `double`, very generous type convertions are done: More or less every value is converted.

    E.g. `"abc"` can be converted to `double`, the result is `NaN` (not-a-number). The boolean value “true” is converted to 1, “false” to 0.

Subtle Differences III

- Let the context node be

  ```
  <E A="3"/>
  ```

  and suppose that the document was not schema-validated, so the attribute is of type `untypedAtomic`.

- Then `1 to @A` works.

  The `untypedAtomic` value is converted to `integer`.

- But `1 to @A+1` gives a type error.

  `+` accepts different numeric types, and the `untypedAtomic` value of `@A` is converted to `double`. But `double` is no legal input type for `to`.

- A type conversion is needed: `1 to xs:integer(@A)+1`.

Contents

Node Properties (1)

- `name([n])`: Node name (string that includes prefix)

  Argument type: `node()?`, result type: `xs:string`. Function returns empty string if the input is the empty sequence or a document, text, or comment node. The argument is optional (default: context node).

- `node-name(n)`: Node name (`QName`: URI, local part)

  Argument type: `node()?`. Result type: `xs:QName?`. Function is new in XPath 2.0.

- `local-name([n])`: Node name (without prefix)

  Argument type: `node()?`, result type: `xs:string`. Argument is optional.

- `namespace-uri([n])`: Namespace part of node name.

  Argument type: `node()?`, result type: `xs:string`. Argument is optional. Result is empty string if node has no namespace.

Node Properties (2)

- `string([n])`: String value of a node or atomic value.

  Argument: `item()?`. Result: `xs:string`. Atomic values are casted to string, nodes are mapped to their string value (see Chapter 9, e.g. for element nodes, this is the concatenation of all decendant text nodes).

- `data(n)`: Replaces nodes in input by typed value.

  Argument: `item()*` (arbitrary sequence). Result: `xs:anyAtomicType*`. This is atomization (see above): Atomic values in the input sequence are copied to the output isequence unchanged, nodes are replaced by their typed value. Nodes with pure element content cause a runtime error if document was schema-validated. New in XPath 2.0.

- `nilled(n)`: True if element contains `xsi:nil="true"`.

  Argument: `node()`. Result: `xs:boolean?`. If the document was not validated (wrt schema), the result is false even if attribute is present. The empty sequence is returned for non-element nodes. New in XPath 2.0.

Node Properties (3)

- `document-uri(`*n*`)`: URI under which the document can be accessed.

  Argument: `node()?`. Result: `xs:anyURI?`. For document nodes *n*, an absolute URI *x* is returned, such that *n*=`doc(`*x*`)`. For other nodes, the empty sequence, or if no such URI is known, the result is the empty sequence. New in XPath 2.0

- `base-uri([`*n*`])`: Base URI for resolving relative URIs.

  Argument: `node()?`. Result: `xs:anyURI?`. Base URI of the node, or if it has none, searches recursively the ancestors. The URI of the input document, an external entity, or of an `xml:base` attribute is returned. If no URI is found, the empty sequence is returned. The argument is option (default: context item). New in XPath 2.0. See also `static-base-uri()` and `resolve-uri()`.

Node Properties (4)

- `lang(`*l*`, [`*n*`])`: Checks whether language *l* is specified with `xml:lang` for node *n*

  Argument *l*: `xs:string` (e.g., `"de"`, `"en-US"`), *n*: `node()` (default: context node). Result: `xs:boolean`. This function first determines the value of the attribute `xml:lang` of node *n* or its nearest ancestor that has such an attribute. This attribute can be found with the following XPath expression: `(ancestor-or-self::*/@xml:lang)[last()]`. If there is no such attribute node, the function returns false. Otherwise, let the value of the attribute be *x*. If *x* and `l` are equal (ignoring case), the result is true. If *l* is the prefix of *x* before the hyphen (again ignoring case), the result is true. Otherwise the result is false. The second argument has been added in XPath 2.0.

Finding Nodes (1)

- `doc(`*u*`)`: Get document node for given URI.

  Argument: `xs:string?` Result: `document-node()?`. A runtime error occurs if there is no document with the given URI. This function is stable, it is guaranteed to return the same node if it is called several times with the same URI (during the evaluation of a query). New in XPath 2.0 (however, XSLT 1.0 has a function `document()`).

- `doc-available(`*u*`)`: Check whether there is a document with a given URI.

  Argument: `xs:string?` Result: `xs:boolean`. This returns `true` if `doc(`*u*`)` would return a node. It can be used in an `if`-expression to avoid the runtime error that `doc(`*u*`)` would generate if there is no document with URI *u*. New in XPath 2.0.

Finding Nodes (2)

- `collection(`*u*`)`: Nodes in container identified by URI.

  Argument: `xs:string?` Result: `node()*`. This might be the document nodes of the documents in a directory identified by the URI. Containers also exist in XML databases. It is not necessary that only document nodes are returned. New in XPath 2.0.

- `root(`*n*`)`: Root of the tree that contains node *n*.

  Argument type: `node()?`, result type: `node()?`. Argument is optional (default: context node). Function is new in XPath 2.0.

Finding Nodes (3)

- `id(`*i*`, [`*n*`])`: Nodes with ID in *i* in document containing node *n*.

  Argument *i*: `xs:string*`, *n*: `node()` (default: context node). Result: `element()*`. Each string in *i* is parsed like an `IDREFS` value, i.e. it might contain several IDs, separated by spaces. All these IDs in all strings in the sequence *i* are considered for a possible match (values that are not syntactically legal IDs are ignored). For each such ID, the (first) element node with that ID in the document containing node *n* is added to the output sequence. It is no error if there is no node with a given ID. The output sequence contains the resulting nodes in document order without duplicates. The root node reachable from node *n* must be a document node. New in XPath 2.0.

Finding Nodes (4)

- `idref(`*i*`, [`*n*`])`: Nodes with IDREF value containing an ID in *i* (in document containing node *n*).

  Argument *i*: `xs:string*`, *n*: `node()` (default: context node). Result: `node()*` (actually, only element and attribute nodes are returned). Candidate IDs are determined from the list *i* as above. Then every attribute and element node in the document identified by *n* that contains an `IDREF`/`IDREFS`-value that matches an ID in the candidate list is returned. In case of `IDREFS`-values, it suffices if one of the IDs matches a candidate ID (from *i*). Note that in the classical DTD case, the attribute node of type `IDREF`/`IDREFS` is returned. Element nodes are returned only for schema-validated documents, when their contents is of this type. Again, the result is a list of nodes in document order without duplicates. New in XPath 2.0.

Contents

Functions for Sequences (1)

- s_1 , s_2: Sequence concatenation.

  The operands and the result have type `item()*` (arbitrary sequences).

- `index-of(`s`,` e`,` $[c]$`)`: Return list of positions at which element e occurs in sequence s (using collation c).

  Argument s: `xs:anyAtomicType*`, e: `xs:anyAtomicType`, c: `xs:string`. Result: `xs:integer*`. Values of type `xs:untypedAtomic` are compared as if they were of type `xs:string`. The collation c is only important for string comparisons. If an element of s cannot be compared with e, it counts as different (no type error occurs). Note that the input sequence is atomized before the comparison (this may change positions). The first element of s has position 1. E.g. `index-of((10,20,30,20), 20) = (2,4)`. New in XPath 2.0.

Functions for Sequences (2)

- `insert-before(`s_1`,` p`,` s_2`)`: Returns the sequence consisting of the prefix of s_1 before position p, then s_2, then the rest of s_1.

  Argument s_1, s_2: `item()*`, p: `xs:integer`. Result: `item()*`. Positions are counted from 1. Since XPath makes no difference between single elements and sequences of length 1, s_2 can also be an element.
  E.g. `insert-before((10,20,30), 2, 15) = (10,15,20,30)`. XPath never does any updates, so s_1 is not changed. If $p \leq 0$, it is treated like $p = 1$. If $p >$ length of s_1, the insertion is done at the end. New in Xpath 2.0.

- `remove(`s`,` p`)`: Returns a copy of sequence s without element at position p.

  Argument s: `item()*`, p: `xs:integer`. Result: `item()*`. The effect is the same as `$s[position() ne $p]`. New in XPath 2.0.

Functions for Sequences (3)

- `subsequence(`*s*`,` *f* `, [`*l*`])`: Returns subsequence of *s* consisting of (at most) *l* elements ("length") starting at position *f* ("from").

  Argument: *s*: `item()*`, *f*: `xs:double`, *l*: `xs:double` (default: infinite). Result: `item()*`. First item is position 1. If *l* is outside the bounds of index positions, it is implictly corrected (no error occurs). The arguments *f* and *l* are rounded to integers. They are declared as `xs:double`, because many computations on untyped data return this type. (Furthermore, it increases the symmetry with `substring`, which existed already in XPath 1.0: There, all numbers were double values.) New in XPath 2.0.

- `reverse(`*s*`)`: Gives *s* with inverse order of elements.

  Argument: *s*: `item()*`, Result: `item()*`. New in XPath 2.0.

Functions for Sequences (4)

- `distinct-values(`*s*`, [`*c*`])`: Returns a sequence that contains the same elements as *s*, but without duplicates (using collation *c* for string comparisons).

  Argument: *s*: `anyAtomicType*`, *c*: `xs:string` (default in static context). Values of type `xs:untypedAtomic` are compared as if they were strings, but they are still of type `xs:untypedAtomic` in the output (they are not converted to `xs:string`). The output order is implementation-dependent (e.g., a typical implementation would be to sort the elements, but some internal order could be used). The implementation is also free to choose any of the equal elements, e.g. if the collation makes `"A" eq "a"`, and both appear in the input, it is not clear which one will appear in the output. Elements of different type that connot be compared with `eq` are considered as different (no type error occurs). Also duplicates of `NaN` are eliminated, although it is usually not considered as equal to itself. New in XPath 2.0.

Exercise

- Suppose that the context node is

```
<x a="c1 c2">
    <y a="c2 c3 c4"/>
    <z a="c2">
        <y a="c2"/>
    </z>
    <y a="c2"/>
</x>
```

  Attribute *a* is declared as `xs:NMTOKENS`.

- What is the result of

```
index-of(//y/@a)
```

Optimizer Hint

- `unordered(s)`: Returns arbitrary permutation of `s`.

  Argument: `item()*`. Result: `item()*`. Note that this cannot be used for e.g. computing a random list element. In many systems, it will simply be the identity mapping. However, it tells the optimizer that the user does not care about the order of the result: Otherwise, XPath nearly always defines an order of the elements, because it works with sequences, not (multi)sets. The query optimizer might then choose a more efficient evaluation strategy for the argument `s` (to some degree, also for outer expressions, but that is more difficult: The typical case is probably to use `unordered` on the outermost level, although one can construct cases where it is more efficient somewhere inside the expression.) Note that `unordered` is unnecessary, when later a function like `count` is applied, for which the exact order is anyway not important. If duplicate elimination is needed, as e.g. for s_1 `|` s_2, enclosing it in `unordered(...)` probably does not help too much (unless the optimizer can prove that there will be no duplicates). New in XPath 2.0.

Contents

Aggregation Functions (1)

- `count(s)`: Number of elements in sequence *s*.

  Argument type: `item()*`. Result: `xs:integer`. This is the length of *s*.

- `sum(s, [z])`: Sum of elements in sequence *s*. For the empty sequence, *z* is returned.

  Argument *s*: `xs:anyAtomicType*`, *z*: `anyAtomicType?` (default: integer 0). Result: `xs:anyAtomicType`. After atomization, XPath determines a common type for the sequence elements (one of: `xs:integer`, `xs:decimal`, `xs:float`, `xs:double`, `xs:dayTimeDuration`, `xs:yearMonthDuration`) and converts all elements to this type with the usual promotion rules (`xs:untypedAtomic` is converted to `xs:double`). If this is not possible, the function raises an error. Otherwise, the sum of the converted values is returned (unless the sequence is empty, in which case *z* is returned: Important for dynamically typed systems: `()` has no type). In XPath 1.0, only the sum of `double`s could be computed (also no *z*).

Aggregation Functions (2)

- `avg(s)`: Average of elements in sequence *s*.

  Argument type: `anyAtomicType*`. Result: `xs:anyAtomicType?`. This first computes the sum of the elements of *s* (see `sum` above), and then divides the result by the number of elements in *s*. If *s* is the empty sequence, the empty sequence is returned.

- `min(s, [c])`: Minimum of elements in sequence *s*.

  Argument *s*: `anyAtomicType*`, *c*: `xs:string` (default: default collation in context). Result: `xs:anyAtomicType?`. The elements of the sequence are first atomized, and then converted to a common type (which must support the `le` operator, e.g. `xs:QName` and `xs:anyURI` are excluded; `xs:untypedAtomic` is converted to `xs:double`. If this is not possible, an error occurs. Then an element is returned that is $\leq$ all other elements. The collation *c* is only important for string types. New in XPath 2.0.

- `max(s, [c])`: Maximum of elements in sequence *s*.

Exercise

- Consider again:

```
<GRADES-DB>
  ...
  <RESULT>
    <SID>101</SID>
    <CAT>H</CAT>
    <ENO>1</ENO>
    <POINTS>10</POINTS>
  ...
```

- What is the average number of points for Homework 1?
- What does this mean?

```
for $p in max(//POINTS)
return //RESULT[POINTS=$p]
```

Boolean Functions (1)

- `true()`: Constant value "true".
- `false()`: Constant value "false".

  Result: `xs:boolean`. Otherwise, XPath has no boolean constants.

- `empty(`*s*`)`: Sequence *s* is empty.

  Argument: `item()*`. Result: `xs:boolean`. If *s* is the empty sequence, the function returns `true`, otherwise, it returns `false`. New in XPath 2.0.

- `exists(`*s*`)`: Sequence *s* is not empty.

  Argument: `item()*`. Result: `xs:boolean`. If *s* is the empty sequence, the function returns `false`, otherwise, it returns `true`. Often, this function is not needed: For a sequence of nodes, the effective boolean value is true iff the sequence is not empty. But if the first element can be an atomic value, `exists()` might be important. New in XPath 2.0.

Contents

Boolean Functions (3)

- `deep-equal(`s_1`, `s_2`, [`c`])`: Check whether s_1 and s_2 are very similar, including descendant nodes.

  Argument s_1, s_2: `item()*`, c: `xs:string` (collation). Two sequences are deep-equal iff they have the same length, and each pair of elements at the same position is deep-equal. Atomic values are deep-equal if they can be compared with `eq` (so they have similar types), and `eq` returns true. Two nodes can be deep-equal only if they have the same kind. Two text nodes are deep-equal if their string-values are equal. Two attribute nodes are deep-equal if they have the same name, and their typed value is deep-equal. Two element nodes are deep-equal if they have the same name, their set of attribute nodes is deep-equal, and: (1) both have a simple type, and their typed values are equal, or (2) (a) both have a complex type with element-only content, or both a complex type with mixed content, or both a complex type with empty content, and (b) their sequences of child nodes (ignoring comment and PI nodes) is deep-equal. New in XPath 2.0. Continued →

Boolean Functions (4)

- `deep-equal(`s_1`, `s_2`, [`c`])`: Continued (comments):
  - If nodes are identical, i.e. n_1 `is` n_2, then also `deep-equal(`n_1`, `n_2`)`. The converse is not true.

    E.g., if one copies a tree, the result is `deep-equal`, but not identical. This also holds if the same subtree appears in two parts of a document: Nodes with different parents can still be deep-equal.
  - If `A` is declared e.g. as `decimal`, the following nodes are deep-equal:

```
<E A="3" B="xyz"/>
<E B="xyz" A="3.0"><!-- comment --></E>
```

  - Whitespace-only text nodes are not ignored in the comparison.

Numeric Functions (1)

- `abs(`x`)`: Absolute value.

  There a four versions of this function: One with argument and result type `xs:integer`, one for the numeric type `xs:decimal`, one for `xs:float`, and one for `xs:double`. If x is negative, the function returns $-x$, otherwise x (so that the result is always ≥ 0). New in XPath 2.0.

- `ceiling(`x`)`: Round to next greater whole number.

  Again, there are four versions of this function for the four important numeric types. The result type is the same as the argument type, e.g. `ceiling(1.2)=2.0`. This function exists already in XPath 1.0, but there all numbers were double precision floating point numbers.

- `floor(`x`)`: Round to next smaller whole number.

  Again, there are four versions for the four important numeric types. The result type is the same as the argument type, e.g. `floor(1.8)=1.0`.

Numeric Functions (2)

- `round(`x`)`: Round to nearest whole number.

  The four numeric types are supported (see above). The result type is the same as the argument type. E.g. `round(1.2)=1.0`, `round(1.8)=2.0`. If x ends in `.5`, it is rounded upwards: `round(1.5)=2.0`, `round(-1.5)=-1.0`.

- `round-half-to-even(`x`, [`n`])`: Round x to n decimal places to the right of the decimal point.

  There are the usual four versions of this function, but the typical case is with argument x: `xs:decimal?` and result `xs:decimal?`. The argument n has always type `xs:integer` (the default value is 0). The function produces the nearest number that is a multiple of 10^{-n}.
  E.g. `round-half-to-even(10.183, 1) = 10.2`. If the input x is exactly in the middle between two possible results, the one with an even last digit is chosen (e.g. `0.5`$\rightarrow$`0`, `1.5`$\rightarrow$`2`). This ensures that rounding does not systematically make the average slightly larger. New in XPath 2.0.

String Functions (1)

- `codepoints-to-string(c)`: Construct string for given sequence of Unicode character codes.

  Argument: `xs:integer*`. Result: `xs:string`. New in XPath 2.0.

- `string-to-codepoints(s)`: Map given string into sequence of Unicode character codes.

  Argument: `xs:string?`. Result: `xs:integer*`. Note that a character that is represented as a surrogate pair (two 16-bit numbers in the internal string representation) counts only as one character and thus results in a single number in the output sequence. The resulting numbers are in the range `1` to `0x10FFFF`. This function is new in XPath 2.0.

String Functions (2)

- `normalize-unicode(`*s*`,[`*f*`])`: Replace different variants to denote a character by a unique representation.

  Argument *s*: `xs:string?` (input string to be normalized), *f*: `xs:string` (normalization form/algorithm, default `"NFC"`). E.g. characters with accents like `ä` can be represented as a single character code, or as two (`a` followed by `&#x0308`: "Combining Diaresis"). Thus, string comparisons might fail although the characters look identical. NFC uses the single, combined character. NFKC in addition maps "compatibility variants" of characters to a single code. It is recommended that XML documents are normalized, therefore these problems usually don't occur. One problem is that NFC permits a combining character at the beginning of a string, therefore the concatenation of two NFC-normalized strings is not necessarily NFC-normalized. The normalization form `"fully-normalized"` would exclude this (e.g. by prepending a space to the lonely combining character). XPath implementations are not required to offer other normalization forms besides `"NFC"`.

String Functions (3)

- `compare(`s_1`, `s_2`, [`c`])`: Returns -1, 0, 1 depending on which string comes first according to collation c.

  Argument s_1, s_2: `xs:string?`, c: `xs:string` (must be URI, default is default collation from static context). Result: `xs:integer?`. The result is -1 if s_1 comes before s_1 (in alphabetic or other order c), 1 if s_2 comes before s_1, and 0 if s_1 and s_2 are equivalent (depending on the collation, e.g. ß might count as equal to `ss`). The function `compare` is implicitly used by the comparison operators for strings (therefore, the results are guaranteed to be compatible).

- `codepoint-equal(`s_1`, `s_2`)`: Strings are equal byte by byte.

  Argument s_1, s_2: `xs:string?`. Result: `xs:boolean?`. This returns true if the two strings are exactly equal.

String Functions (4)

- `concat(`s_1`, `s_2`, ..., `s_n`)`: Concatenation of s_1 to s_n.

  This is the only function with a completely variable number $n \geq 2$ of arguments (retained for compatibility with XPath 1,0). All other functions have only a fixed number of versions that differ in the number of arguments (or the specific numeric type). The arguments have type `xs:anyAtomicType?`. They are converted to `xs:string` before the concatenation (the empty sequence is treated as empty string).
  The result has type `xs:string`.

- `string-join(`s`,`d`)`: Returns the concatenation of the strings in sequence s, separated by delimiter d.

  Argument s: `xs:string*`, d: `xs:string`. Result: `xs:string`.
  E.g. `string-join(("a", "bc", "d"), ", ")` gives `"a, bc, d"`.
  New in XPath 2.0.

String Functions (5)

- `string-length([s])`: Number of characters in *s*.

  Argument: `xs:string?` (default: string value of context item). Result: `xs:integer`. The string length of the empty sequence is 0. Note that a surrogate pair (used for code points above `0xFFFF`) counts as one character, not two.

- `substring(s, f, [l])`: Returns the substring of *s* that starts at position *f* and consists of *l* characters.

  Argument *s*: `xs:string?` (input string), *f*: `xs:double` (from position), *l*: `xs:double` (maximal length of output, default: infinite). The first character has position 1. E.g. `substring("abcde", 2, 3)` is `"bcd"`. The numbers *f* and *l* are rounded. If *f* is 0 or negative, it is implicitly replaced by 1. In the two-argument from, when gets the entire rest of the input string starting at position *f*.

String Functions (6)

- `normalize-space([s])`: Remove leading and trailing whitespace, replace internal sequences of whitespace characters by a single ' '.

  Argument: `xs:string?` (default: string value of context item). Result: `xs:string`. This function has the same effect as `whiteSpace="collapse"` in XML Schema.

- `translate(s, a, b)`: Maps every character in *s* that appears in *a* to the corresponding character in *b*.

  Argument *s*: `xs:string?`, *a*, *b*: `xs:string`. Result: `xs:string`. Every character in *s* that appears in *a* at position *i* is replaced by the character at position *i* in *b*. If *b* is shorter than *i*, the character is deleted. Characters in *s* that do not appear in *a* are copied to the output string unchanged. Example: `translate("aBacx", "abc", "AB")` gives `"ABAx"`.

String Functions (7)

- `upper-case(s)`: Make all letters upper case.

  Argument: `xs:string?`. Result: `xs:string`. Note that the string length may change, e.g. `ß` is mapped to `SS`. Some national conventions in certain countries are not respected, if necessary, use `replace`. New in XPath 2.0.

- `lower-case(s)`: Make all letters lower case.

  Argument: `xs:string?`. Result: `xs:string`. New in XPath 2.0.

String Functions (8)

- `contains(`s_1`,` s_2`,` $[c]$`)`: Check whether s_2 appears as a substring in s_1.

  Arguments s_1, s_2: `xs:string?`, c: `xs:string` (identifies collation, this argument is new in XPath 2.0). Result: `xs:boolean`. The collation defines a way to map a string to a sequence of "collation units", then `true` is returned if this sequence for s_2 is a subsequence of the sequence for s_1. E.g. `contains("Straße", "s", "http://...")` might return true if the referenced collation maps "ß" to two collations units corresponding to `ss`. Also the converse case is possible: Several input characters may be mapped to a single collation unit, in which case the substring test with only one of these characters would fail. Finally, there can be "ignorable collation units", which are deleted for both strings before the subsequence test. There can be collations that do not support the mapping to collation units (since for normal comparisons, this feature is not needed). Then an error may be raised. If s_2 is empty or the empty sequence, the result is `true`.

String Functions (9)

- `starts-with(`s_1`,` s_2`,` $[c]$`)`: Check whether s_2 is prefix of s_1.

  Arguments s_1, s_2: `xs:string?`, c: `xs:string` (identifies collation, this argument is new in XPath 2.0). Result: `xs:boolean`.

- `ends-with(`s_1`,` s_2`,` $[c]$`)`: Check whether s_2 is suffix of s_1.

  Arguments s_1, s_2: `xs:string?`, c: `xs:string` (identifies collation, this argument is new in XPath 2.0). Result: `xs:boolean`.

String Functions (10)

- `substring-before(`s_1`, `s_2`, [`c`])`: Return the prefix of s_1 before the first match of s_2.

  Arguments s_1, s_2: `xs:string?`, c: `xs:string` (identifies collation, this argument is new in XPath 2.0). Result: `xs:string`. A "minimal match" is used. E.g. if "`-`" is ignorable, `substring-before("a-b", "-b", ...)` is `"a-"`, because `"-b"` matches `"b"`. If there is no match, the result is the empty string.

- `substring-after(`s_1`, `s_2`, [`c`])`: Return the suffix of s_1 after the first match of s_2.

  Arguments s_1, s_2: `xs:string?`, c: `xs:string` (identifies collation, this argument is new in XPath 2.0). Result: `xs:string`. If s_2 is the empty string, the first match is at the beginning, thus the entire string s_1 is returned. If there is no match, the result is the empty string.

Regular Expressions (1)

- `matches(`*s*`, `*p*`, [`*f*`])`: Checks whether (a substring of) *s* matches pattern *p* (considering flags *f*).

  Argument *s*: `xs:string?`, *p*: `xs:string`, *f*: `xs:string`. Result: `xs:boolean`. Basically, the regular expression syntax is the same as for XML Schema, however, there are a few additions: Since normally a match can occur anywhere inside *s*, `^` and `$` are supported: `^` matches only at the beginning of the string, or at the beginning of a line if flag `m` (multi-line mode) is used. `$` matches at the end. Quantifiers like `*?` are supported, which means that the shortest possible match is taken. Groups in parentheses `(...)` may be referenced with a construct of the form `\`*n*, e.g. `\1`. The flag `s` ("single line mode") means that "`.`"Âămatches also newline, otherwise "`.`" matches only all characters except newline. The flag `i` makes comparisons case-insensitive. The flag `x` removes all whitespace from *p* except inside character classes `[...]` (permits to split a regular expression into several lines). New in Xpath 2.0

Regular Expressions (2)

- `replace(`*s*`, `*p*`, `*r*`, [`*f*`])`: Replaces all non-overlapping occurrences of pattern *p* in *s* by *r* (with flags *f*).

  Argument *s*: `xs:string?`, *p*, *r*, *f*: `xs:string`. Result: `xs:string`. If two matches overlap, the first one is used. Matches for parenthesized subexpressions of *p* can be used in *r* with "variables" `$`*n*. If several cases of an alternative `|` match at the same position, the first one is used. If subexpression *n* was not used in the match, `$`*n*`=""`. Patterns that match the empty string are forbidden. In *r*, the character `$` must be written `\$`, and `\` as `\\`. New in XPath 2.0.

- `tokenize(`*s*`, `*p*`, [`*f*`])`: Splits *s* into substrings seperated by parts that match pattern *p* (with flags *f*).

  Argument *s*: `xs:string?`, *p*, *r*, *f*: `xs:string`. Result: `xs:string*`. E.g., `tokenize("ab c  def ", "\s+")` yields `("ab", "c", "def", "")` (note: `\s` matches ' ', `TAB`, `CR`, `LF`). *p* must not match `""`. New in XPath 2.0.

Exercise

- Consider again:

```
<GRADES-DB>
  <STUDENT>
    <SID>104</SID>
    <FIRST>Maria</FIRST>
    <LAST>Brown</LAST>
  </STUDENT>
  ...
  <RESULT>
    <SID>101</SID>
    <CAT>H</CAT>
  ...
```

- Print first and last name of all students who did not submit any homework.

Contents

Context Functions (1)

- `last()`: Context size (from dynamic context/focus).

  Result type: `xs:integer`. Returns the length of the sequence that is currently being processed (see above).

- `position()`: Context position.

  Result type: `xs:integer`. Position (counted from 1) of the current context item in the sequence that is currently being processed.

- `static-base-uri()`: Base URI from static context.

  Result type: `xs:anyURI?`. This could e.g. be the URI of the XSLT stylesheet. New in XPath 2.0.

- `default-collation()`: Sort order for strings.

  Result type: `xs:string`. New in XPath 2.0.

Context Functions (2)

- `current-dateTime()`: Current date and time.

  Result type: `xs:dateTime`. This is stable, i.e. it does not change during the evaluation of a single query. New in XPath 2.0.

- `current-date()`: Current date.

  Result type: `xs:date`. This is simply the date component (with timezone) of the value returned by `current-dateTime()`. New in XPath 2.0.

- `current-time()`: Current time.

  Result type: `xs:time`. This is the time component (with timezone) of the value returned by `current-dateTime()`. New in XPath 2.0.

- `implicit-timezone()`: Timezone used for local time.

  Result type: `xs:dayTimeDuration`. New in XPath 2.0.

URI Utility Functions (1)

- `resolve-uri(`*r*`, [`*b*`])`: Relative URI → absolute URI.

  Argument *r*: `xs:string?` (relative URI), *b*: `xs:string` (base URI, default: base URI from static context). Result: `xs:anyURI?`. If *r* is already an absolute URI, it is returned unchanged. New in XPath 2.0.

- `escape-uri(`*s*`, `*r*`)`: Escape special characters as *%XY*.

  Argument *s*: `xs:string` (URI in unescaped form), *r*: `xs:boolean` ("escape reserved", see below). Result: `xs:string`. Letters, digits, and `-`, `_`, `.`, `!`, `~`, `*`, `'`, `(`, `)`, and `%` are not escaped. If *r* is true, all other characters are escaped (e.g. also `/`). If *r* is false, the characters `;`, `/`, `?`, `:`, `@`, `&`, `=`, `+`, `$`, `,`, `[`, `]`, and `#` are not escaped. Note that `%` is actually a reserved character, but this function does not touch it in order to support "partially escaped" input strings (and make this function idempotent). If necessary, use `replace()`. New in XPath 2.0.

URI Utility Functions (2)

- `encode-for-uri(`*u*`)`: Encodes file/directory name.

  Argument: `xs:string?`. Result: `xs:string`. All characters except ASCII letters `a-z` and `A-Z`, digits `0-9`, `-`, `_`, `.`, and `~` are encoded as *%XY*. Note that e.g. also "`/`" is encoded. New in XPath 2.0.

- `escape-html-uri(`*u*`)`: Encode non-ASCII characters in UTF-8 and then escape them as *%XY*.

  Argument: `xs:string?`. Result: `xs:string`. All characters with codes outside the range 32 to 126 are translated in a way appropriate for web browsers. See HTML 4.0 spec., Appendix B.2.1. New in XPath 2.0.

- `iri-to-uri(`*u*`)`: Internationalized URI (IRI) $\rightarrow$ URI.

  Argument: `xs:string?`. Result: `xs:string`. Translates characters not valid in an URI to UTF-8, then *%XY*-encodes the bytes. May use special encoding for domain names. See RFC 3987, 3.1. New in XPath 2.0.

Namespaces

- `in-scope-prefixes(`*n*`)`: Return a list of namespace prefixes that a declared for a given element node.

  Argument *n*: `element()`. Result: `xs:string()*`. This function returns all namespace prefixes that are declared in node *n* or one of its ancestors. The order os the prefixes is not prescribed. An empty string corresponds to the default namespace. The prefix "`xml`" is always contained in the result. This function is new in XPath 2.0. It is a replacement for the namespace axis, which should no longer be used for efficiency reasons.

- `namespace-uri-for-prefix(`*p*`, `*n*`)`: URI of namespace with prefix *p* as valid for node *n*.

  Argument *p*: `xs:string`, *n*: `element()`. Result: `xs:string?`. The empty sequence is returned if no namespace declaration for prefix *p* is found. New in XPath 2.0.

QNames

- `local-name-from-QName(`*n*`)`: Local part of QName.

  Argument *n*: `xs:QName?`. Result: `xs:string?`. New in XPath 2.0.

- `namespace-uri-from-QName(`*n*`)`:
  Returns the namespace URI of QName *n*.

  Argument *n*: `xs:QName?`. Result: `xs:string()?`. An empty sequence is returned for the empty sequence as input, and if the input QName is in no namespace. New in XPath 2.0.

- `expanded-QName(`*u*`, `*n*`)`: Constructs QName from namespace URI *u* and local name *n*.

  Argument *u*: `xs:string?`, *n*: `xs:string`. Result: `xs:QName`. If the first argument is the empty sequence or the empty string, the result is in no namespace. New in XPath 2.0.

Error and Trace Functions

- `error([e], [m], [x])`: Terminates execution, *e*, *m*, *x* are used for generating an error message.

  Argument *e*: `xs:QName` (identifier for error), *m*: `xs:string` (description of error), *x*: `item()*` (additional data, error object). In the two and three argument versions, *e* may be the empty sequence. Result: Does not return. The exact form of the error message is implementation dependent. New in XPath 2.0.

- `trace(x, m)`: Prints data *x* to a trace file labelled by message *m*, returns *x*.

  Argument *x*: `item()*`, *m*: `xs:string`. This is the identity mapping on the first argument, but with the side effect to insert it (together with message *m*) into the trace data set (e.g., trace file). Note that one cannot rely on any specific order of the entries. New in XPath 2.0.

References

- Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon (Editors): XML Path Language (XPath) 2.0. W3C Recommendation, 23 January 2007. [http://www.w3.org/TR/xpath20/]
- Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, Norman Walsh (Ed.): XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, 23 Jan. 2007, [http://www.w3.org/TR/xpath-datamodel/]
- Ashok Malhotra, Jim Melton, Norman Walsh (Ed.): XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, 23 January 2007. [http://www.w3.org/TR/xpath-functions/]
- G. Ken Holman: Definitive XSLT and XPath. Prentice Hall, 2002, ISBN 0-13-065196-6, 373 pages.
- Michael Kay: XPath 2.0 Programmer's Reference. Wiley/Wrox, 2004, ISBN 0-7645-6910-4, 552 pages.
- Michael Kay: XSLT 2.0 Programmer's Reference, 3rd Edition. Wiley/Wrox, 2004, ISBN 0-7645-6909-0, 911 pages.
- Miloslav Nic, Jiri Jirat: XPath Tutorial. Zvon [http://www.zvon.org/xxl/XPathTutorial/General/examples.html]