software AG

Tamino

X-Query Reference Guide

Version 10.7

May 2021

WEBMETHODS

This document applies to Tamino Version 10.7 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: INS-XQL-REF-107-20210510

Table of Contents

Preface

Caution: X-Query is deprecated. This is, it might still be used, but without warranty. Errros will not be corrected anymore.

This document is the reference manual for the X-Query language and describes how each language construct is defined and used in the current version of Tamino. If there is an XPath equivalent, you will find a link that points to the description of the corresponding item in the XPath specification.

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at **https://documentation.softwareag.com**.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at **https://empower.softwareag.com/**.

You can find product information on the Software AG Empower Product Support website at **https://empower.softwareag.com**.

To submit feature/enhancement requests, get information about product availability, and download products, go to **Products**.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the **Knowledge Center**.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at **https://empower.softwareag.com/public_directory.aspx** and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at **http://techcommunity.softwareag.com**. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 Expressions

This chapter describes the expressions available in X-Query. The names of the expressions correspond to the language constructs of X-Query. In many cases, these are equivalent to an XPath expression of the same name. However, in a few cases the meaning is different or the expression only exists in X-Query, since its serves a special purpose in the context of a database. A syntax diagram illustrates the definition as it appears in the respective production rule of the X-Query grammar. The description is followed by an XPath compatibility note and one or more examples.

The most basic parts of the X-Query language are not covered; we summarize them here. Digits, numbers, and operators are defined as in the lexical rules of the XPath specification (in **Section 3.7**). However, X-Query adds to the list of operators the special contains operator ~=, which provides text retrieval capabilities. The list of operator names (`OperatorName`, defined in XPath, **Section 3.7, Rule 33**) is extended by `adj`, `after`, `before`, `betw`, `between`, `near`, `intersect`. All of these additions are described below.

The following table lists the expressions available in X-Query, along with the corresponding XPath expressions. An asterisk (*) after the name of an XPath expression indicates that this expression is implemented differently in X-Query. The link in the XPath column leads you to the expression's production rule in the XPath specification.

X-Query Expression	XPath Expression	Short Description
`AbbrevAbsoluteLocPath`	`AbbreviatedAbsoluteLocationPath`	select all nodes satisfying some condition
`AbbreviatedAxisSpecifier`	`AbbreviatedAxisSpecifier`	specify either child or attribute axis
`AbbreviatedStep`	`AbbreviatedStep`	select context node or its parent node
`AbsoluteLocationPath`	`AbsoluteLocationPath`	location path starting at the root node
`AdditiveExpr`	`AdditiveExpr`	add or subtract numeric values
`AndExpr`	`AndExpr`	check if two or more conditions are all true
`Argument`	`Argument`	input to a `FunctionCall`
`AxisName`	`AxisName` *	specify axis by name
`AxisSpecifier`	`AxisSpecifier`	specify axis in a location step
`BetweenExpr`	—	select nodes based on a value range
`EqualityExpr`	`EqualityExpr` *	check if two values are equal
`Expr`	`Expr`	general X-Query expression
`FilterExpr`	`FilterExpr` *	select a subset of nodes satisfying some condition
`FunctionCall`	`FunctionCall`	call to a function
`LocationPath`	`LocationPath`	select nodes along a path

X-Query Expression	XPath Expression	Short Description
MultiplicativeExpr	MultiplicativeExpr	perform multiplication, division and modulo operations
NameTest	NameTest	check for name of specified node
NodeTest	NodeTest *	check for type or name of specified node
NodeType	NodeType *	token representing a type of node
OrExpr	OrExpr	check if one of the given conditions is true
PathExpr	PathExpr	select a set of nodes
Predicate	Predicate	select a subset of nodes for which the predicate is true
PredicateExpr	PredicateExpr	expression used in a predicate
PrimaryExpr	PrimaryExpr	expression without operators
ProximityExpr	—	specify adjacent values
RelationalExpr	RelationalExpr	compare two numeric values
RelativeLocationPath	RelativeLocationPath	select nodes relative to the context node
SequenceExpr	—	select nodes based on sibling positioning
SetExpr	UnionExpr *	select union or intersection of node sets
SortByClause	—	sort a node set
SortByCharacteristics	—	determine sorting order
Step	Step	select a subset of nodes on a given axis
UnaryExpr	UnaryExpr	change number sign

AbbrevAbsoluteLocPath

Select nodes on a location path starting at the root node.

Syntax

AbbrevAbsoluteLocPath

Description

Select nodes on the `RelativeLocationPath` starting at the document root, which is indicated by the initial `//`.

Note: If you know the position of the node(s) to be selected in a document tree, the query will generally run faster. Example: If you want to know which type of medication has been used for the patients in the current database you need not use `//therapy/type`, but can use `/patient/therapy/type` instead.

Compatibility

It corresponds to the expression `AbbreviatedAbsoluteLocationPath` defined in XPath, **Section 2.5, Rule 10**.

Examples

- Select all doctors who discharged patients:

```
//discharged/doctor
```

- Select all persons whose pager numbers begin with a "3":

```
//name[../@pager ~= '3*']
```

- Select all element nodes that are not immediate children of the root element node:

```
/*//*
```

Related Expression

RelativeLocationPath

AbbreviatedAxisSpecifier

Indicates the axis to be used for selecting nodes in a path expression.

Syntax

AbbreviatedAxisSpecifier

Description

If empty, `AbbreviatedAxisSpecifier` indicates the child axis. If it is `@`, it indicates the attribute axis.

Compatibility

It corresponds to the expression `AbbreviatedAxisSpecifier` defined in XPath, **Section 2.5, Rule 13**.

Examples

- Select the grade of a patient's next of kin. The last step of the location path contains the abbreviated axis specifier `@` plus the name of an attribute node, which would be `./attribute::grade` in the unabbreviated syntax.

```
/patient/nextofkin/@grade
```

- Select the name of a patient's next of kin. The last step of the location path contains the empty abbreviated axis specifier plus the name of a child node, which would be `./child::name` in the unabbreviated syntax.

```
/patient/nextofkin/name
```

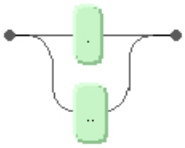
Related Expressions

AxisSpecifier RelativeLocationPath Step

AbbreviatedStep

Select the context node or its parent node.

Syntax



AbbreviatedStep

Description

You can use an `AbbreviatedStep` to indicate either the context node by using the symbol "." or the parent node of the context node by using the symbol "..". The symbol "." is short for `self::node()`, and ".." is short for `parent::node()`.

Compatibility

It corresponds to the expression `AbbreviatedStep` defined in XPath, **Section 2.5, Rule 12**.

Examples

- Select all DCI diagnoses (select all `diagnosis` elements of the current context node that have a value of "DCI"):

```
//diagnosis[.='DCI']
```

- Select all diagnoses that have a headache as symptom (select all `diagnosis` elements whose parent node has the element `symptoms` containing the value "headache"):

```
//diagnosis[../symptoms ~= 'headache']
```

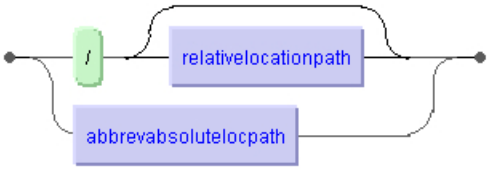
Related Expression

Step

AbsoluteLocationPath

Select nodes on an absolute location path.

Syntax



AbsoluteLocationPath

Description

An `AbsoluteLocationPath` selects nodes on the absolute location path starting at the document root. It consists of "/" optionally followed by a relative location path. The operator "/" by itself selects the root node of the document. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document. Alternatively you can use an abbreviated absolute location path.

Compatibility

It corresponds to the expression `AbsoluteLocationPath` defined in XPath, **Section 2, Rule 2**.

Examples

- Select all patients (the child element node of the root node, which is also the doctype element):

```
/patient
```

- Select all types of medication used (all `type` nodes that are descendants of a `therapy` node of the context node `patient`):

```
/patient/therapy//type
```

- Select all patients whose names start with "B" (all `patient` nodes that satisfy the following predicate expression: `surname` child nodes of the `name` nodes that contain words beginning with "B" or "b"):

```
/patient[./name/surname ~= 'B*']
```

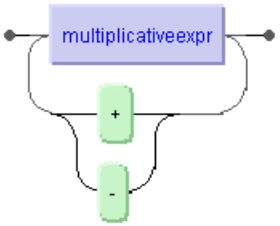
Related Expression

`RelativeLocationPath`

AdditiveExpr

Add or subtract numerical values.

Syntax



AdditiveExpr

Description

An `AdditiveExpr` consists of either a simple `MultiplicativeExpr` or of two or more `MultiplicativeExpr` expressions that are combined by the operators `+` or `-`.

Compatibility

It corresponds to the expression `AdditiveExpr` defined in XPath, **Section 3.5, Rule 25**.

Examples

- Select all patients born after 1959 (all `patient` nodes for which the numeric value of the `born` element +1 is greater than 1960):

```
/patient[born + 1 > 1960]
```

- Select all patients older than 40 years (all `patient` nodes for which the difference between 2001 and the numeric value of the `born` element is greater than 40):

```
/patient[(2001 - ./born) > 40]
```

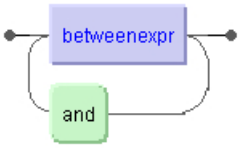
Related Expression

`RelationalExpr`

AndExpr

Check if one or more Boolean operands are all true.

Syntax



AndExpr

Description

If all of the operands evaluate to the Boolean value "true", then "true" is returned, otherwise "false".

Compatibility

The operator name must be specified using lower case.

It corresponds to the expression `AndExpr` defined in XPath, **Section 3.4, Rule 22**, but is extended by `BetweenExpr`. This allows `SequenceExpr` and `ProximityExpr` to be used as operands in an `AndExpr`.

Examples

- Select all patients born in 1950 and living in Bradford (all `patient` nodes that satisfy the following predicate expression: there is an immediate child node `born` whose numeric value equals 1950 and there is also a descendant child node `city` whose string value equals "Bradford"):

```
/patient[born = 1950 and .//city = 'Bradford']
```

- Select all patients born in 1950 having a phone number:

```
/patient[born = 1950 and (address/* after postcode)[position() != last()]]
```

This query demonstrates that you can use a filtered `SequenceExpr` inside an `AndExpr`. It effectively selects all `patient` nodes for which all the following conditions must be met:

- it has a child element node `born` whose numeric value equals 1950
- it has a child element node `address`
- the `address` node has at least three child element nodes that must satisfy these conditions:
  - one of these child nodes has the name `postcode`
  - a sibling node of `postcode` must exist whose position is not the last

There are six possible child nodes for `address` which in document order are `street`, `housenumber`, `city`, `postcode`, `country`, and `phone`. Because of the second predicate expression there must be at least two nodes following a `postcode` node, but if this is true, then it can only be a `phone` node.

Naturally you can write such a condition much more simply:

```
/patient[born = 1950 and address/phone]
```

This query is somewhat faster, since Tamino does not have to expand `address/*` (a `NameTest`) and there is no need to call two functions used in the nested predicate expression. The second condition is evaluated to "true" if an element node `phone` exists that is a child element of `address` which in turn is a child element of a `patient` node.

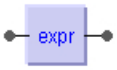
Related Functions

`BetweenExpr`

Argument

The value to be used as input to a function call.

Syntax



Argument

Description

An `Argument` can be any expression which is then used as input to a `FunctionCall`.

Compatibility

It corresponds to the expression `Argument` defined in XPath, **Section 3.2, Rule 17**.

Examples

- Get the number of patients living in Bradford (the argument for `count()` is the node set containing `patient` nodes that satisfy the following filter expression: there are `address` child nodes that have a `city` child node whose string value equals "Bradford"):

```
count(/patient[address/city = 'Bradford'])
```

- Get the number of attributes of the `type` node (the argument for `count()` is the node-set consisting of all attribute nodes that are attached to descendant `type` element nodes):

```
count(//type/@*)
```

- Select all patients without close relatives (all `patient` nodes that do not have `nextofkin` child nodes):

```
/patient[not(nextofkin)]
```

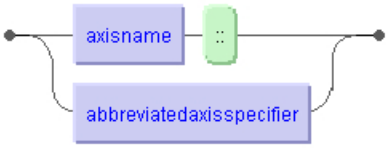
Related Expression

`FunctionCall`

AxisSpecifier

The name of an axis or an abbreviation for it.

Syntax



AxisSpecifier

Description

In a location step, an `AxisSpecifier` determines the initial node set originating in the context node. In practice it means that you navigate through a document tree in a certain direction that is specified by the axis. In the unabbreviated XPath syntax, you can use an `AxisName` to determine the direction of navigation (for example `child::` selects the axis of child nodes in document order starting at the context node).

Since unabbreviated axis specifiers are not supported in X-Query, you can alternatively use an `AbbreviatedAxisSpecifier` to select the `child` axis (`AbbreviatedAxisSpecifier` is empty) or the `attribute` axis (`AbbreviatedAxisSpecifier` is "@"). You can further use a `SequenceExpr` to simulate navigation on the axes `following-sibling` and `preceding-sibling`.

Compatibility

It corresponds to the expression `AxisSpecifier` defined in XPath, **Section 2.1, Rule 5**, with the restriction that `AxisName` is not supported in X-Query.

Example

Select all medication that has been administered in the form of tablets (starting from the root node select all descendant `type` nodes that satisfy the following predicate expression: the value of the `form` node on the attribute axis equals the string value "tablet"):

```
//type[@form = 'tablet']
```

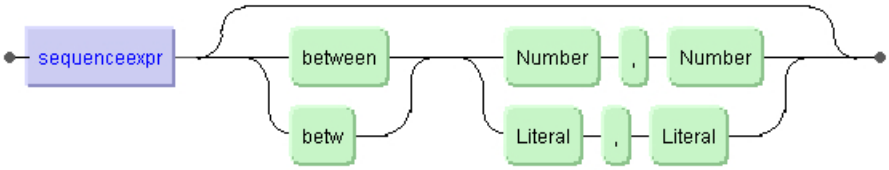
Related Expression

Step

BetweenExpr

Select nodes based on a value range.

Syntax



BetweenExpr

Description

You can select nodes based on a value range, with the limiting values separated by a comma. The range is inclusive of the limiting values. If the first value is larger than the second, the values are implicitly exchanged. The comparison depends on the datatype of the node. If you compare a string value (`Literal`) with a string range, then a lexical comparison is performed using the lexicographical order. If you compare a string value with a numeric range or a numeric value with a string range, then any string values are implicitly converted to numbers and a numeric comparison takes place. If the string cannot be converted to a numeric value, then the conversion yields the special value "NaN" and the comparison fails.

You can think of `BetweenExpr` as a convenient abbreviation for logical conjunctions of `RelationalExpr`, see the compatibility note for an example.

You can use `betw` as a shorthand for `between`. However, both variants must be input in lower case.

Compatibility

There is no `BetweenExpr` defined in XPath. For numeric values, you can simulate this by using relational expressions. The second example below could be rewritten as:

```
//doctor[@pager >= 2 and @pager <= 5]
```

As `RelationalExpr` restricts comparison to numeric values, you cannot in general formulate an equivalent XPath expression for the first example.

`Number` and `Literal` correspond to the respective XPath expressions, as defined in the **Section 3.7**, Rules 30 and 29.

Examples

- Select all doctors whose pager numbers start with "2", "3", "4" (starting from the root node select all descendant `doctor` nodes that satisfy the following predicate expression: the string value of the `pager` attribute is lexicographically ordered between "2" and "5"):

  ```
  //doctor[@pager between '2','5']
  ```

- Select all doctors whose pager numbers are between 2 and 5 (all `doctor` nodes that satisfy the following predicate expression: the string value of the `pager` attribute implicitly converted to a number falls into the interval [2,5]). With our example data, this yields an empty node-set.

  ```
  //doctor[@pager between 2,5]
  ```

- Select all doctors whose pager numbers are between 2 and 5 (all `doctor` nodes that satisfy the following predicate expression: the string value of the `pager` attribute explicitly converted to a number falls into the interval [2,5]). As before, this yields an empty node-set.

  ```
  //doctor[number(@pager) between 2,5]
  ```

- Select all patients born in the 1950's (all `patient` nodes that satisfy the following predicate expression: there is an immediate child element `born` whose numeric value is between 1950 and 1959):

  ```
  /patient[born between 1959,1950]
  ```

Tip: Use `BetweenExpr` instead of `AndExpr` where possible. Usually a query such as `//patient[born between 1950, 1953]` is much more efficient than the equivalent query expression `//patient[born > 1949 and born < 1954]`.

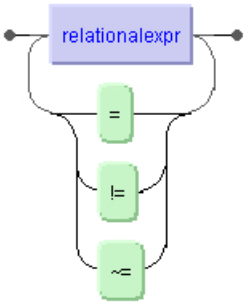
Related Expressions

`SequenceExpr`

EqualityExpr

Check whether two values are equal or not.

Syntax



EqualityExpr

Description

By using the operators = and != you can check if two values are equal or not. In addition, you can test with the help of the contains operator ~= if the left operand contains the word pattern specified by the right operand. The result of the comparison is always one of the Boolean values "true" and "false". If there is a collation defined for either of the operands or if both operands have the same collation defined, then the comparison is based on this collation, otherwise it is character-based (this does not apply for the operator ~=).

When using the white space-separated tokenizer (default), there are also rules for handling character variants such as umlauts or accented characters. As an example the French "é" and "è" will be mapped to "e", before comparison takes place. The German umlauts "ä", "ö", and "ü" will be mapped to "ae", "oe" and "ue" respectively. For a complete and detailed description of how characters are mapped by default and how you can customize this behaviour, please see the section Unicode and Text Retrieval.

Wildcards

With the contains operator you can perform text retrieval including the use of a wildcard character: the value of the right operand is searched word by word regardless of case. The result is "true", if it could be found in the node's value irrespective of its location. A word consists of a non-empty sequence of characters. A wildcard character matches zero or more characters *in a word* so that a

single "*" represents a single word. If the value of the right operand contains more than one word such as in the expression `[node ~= "word1 word2"]` then it is treated as `[node ~= "word1" adj "word2"]`.

The wildcard character is always the asterisk "*" (Unicode value `U+002A`).

Compatibility

The equality operators = and `!=` correspond to the expression `EqualityExpr` defined in XPath, **Section 3.4, Rule 23**. For the additional X-Query operator ~=, there is no equivalent in XPath.

Examples

The examples below are distinguished according to the tokenizer used. For a given database, you can set the tokenizer using the Tamino Manager.

Default (white space separated) Tokenizer

- Select all patients born in 1950 (all `patient` nodes that satisfy the following predicate expression: the value of the `born` child node equals the numerical value 1950):

```
/patient[born = 1950]
```

- Select all patients not born in 1950 (all `patient` nodes that satisfy the following predicate expression: the value of the `born` child node is not equal to the numerical value 1950):

```
/patient[born != 1950]
```

- Select all patients born before 1959:

```
/patient[born < 1959]
```

- Select all patients doing a professional job (all `patient` nodes that satisfy the following predicate expression: the value `occupation` child node contains any case-insensitive form of "Professional"):

```
/patient[occupation ~= 'Professional']
```

- Select all patients doing a professional job (all `patient` nodes that satisfy the following predicate expression: the value `occupation` child node contains any case-insensitive form of "Professional" which is followed by a single word):

```
/patient[occupation ~= 'Professional *']
```

Note: This predicate expression is equivalent with `/patient[occupation ~= 'Professional' adj '*']`.

- Select all patients whose surname contains a word beginning with "At":

```
/patient/name[surname ~= 'At*']
```

- Select all patients whose surname contains a word ending with "ins":

```
/patient/name[surname ~= '*ins']
```

- Select all patients whose surname contains a word with the sequence "ins":

```
/patient/name[surname ~= '*ins*']
```

Japanese Tokenizer

Examples using the Japanese tokenizer are available in the section Pattern Matching in the *XQuery User Guide*.

Related Expression

`RelationalExpr`

Expr

Represents a top-level query expression.

Syntax

Expr

Description

An `Expr` is the top-level expression which is defined in terms of an `OrExpr`. This means that every `Expr` is an `OrExpr` and vice versa.

Compatibility

It corresponds to the expression `Expr` defined in XPath, **Section 3.1, Rule 14**.

Examples

All examples in the expression reference section use expressions in the sense of `Expr`.

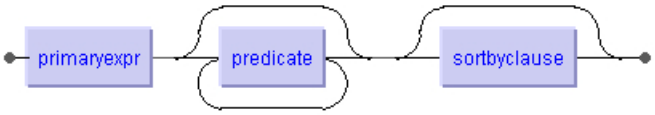
Related Expressions

`OrExpr Argument`

FilterExpr

Select a subset of a given node-set by checking whether they satisfy one or more predicates.

Syntax



FilterExpr

Description

A `FilterExpr` contains some `PrimaryExpr` which constitutes a node-set, and optionally a predicate which filters the node-set according to the predicate expression.

Compatibility

It corresponds to the expression `FilterExpr` defined in XPath, **Section 3.3, Rule 20**.

Example

Select all patients and relatives sharing the surname "Atkins" (a `SetExpr` filtered by a `Predicate` that selects all those names containing a case-insensitive form of "Atkins"):

```
(/patient/name | /patient/nextofkin/name)[surname ~= 'Atkins']
```

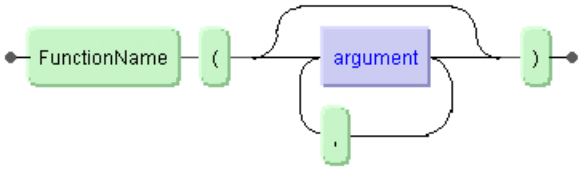
Related Expressions

`Predicate` `PrimaryExpr`

FunctionCall

Call either a built-in function or a user-defined function.

Syntax



FunctionCall

Description

A `FunctionCall` expression is evaluated by using the `FunctionName` to identify a function in the expression evaluation context function library, evaluating each of the arguments, converting each argument to the type required by the function, and finally calling the function, passing it the converted arguments. The result of the `FunctionCall` expression is the result returned by the expression.

A `FunctionName` can be any qualified name (see definition of `QName` in the XML Namespaces specification, **Section 3, Rule 6**) with the exception of those names that are used for a `NodeType`. Standard core functions in XPath do not use a namespace prefix, but Tamino-internal functions use the prefix `ino` such as `ino:id()` or `ino:explain()`. You can also call a function that has been defined as a server extension in the same way as every other function. However, user-defined query functions are only available in those databases in which they had been installed as a server extension.

Compatibility

It corresponds to the expression `FunctionCall` defined in XPath, **Section 3.2, Rule 16**.

You can extend the set of builtin query functions by user-defined functions that are registered in a Tamino database as server extensions (see Tamino X-Tension for more information).

Examples

- Call the Boolean function that always returns "true":

```
true()
```

- Get the number of patients living in Bradford:

```
count(/patient[address/city = 'Bradford'])
```

- Get the query explanation plan for selecting all patients living in cities whose names begin with "B" or "b":

```
ino:explain(/patient//city[. ~= 'B*'])
```

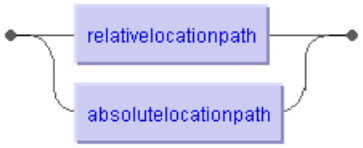
Related Expression

PrimaryExpr

LocationPath

Select nodes along a relative or absolute path.

Syntax



LocationPath

Description

A `LocationPath` selects a set of nodes either relative to the context node if it is a `RelativeLocationPath` or relative to the root node if it is an `AbsoluteLocationPath`. The result of evaluating a location path expression is the node set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes.

Compatibility

It corresponds to the expression `LocationPath` defined in XPath, **Section 2, Rule 1**.

Examples

- Select all patients (using a relative location path select all `patient` nodes starting from the context node which is the same as the collection's doctype element):

```
patient
```

- Select all male patients sorted by their first names (using an absolute location path select all `patient` nodes that have a child element node `sex` whose value matches the pattern "male"; sort this node-set by the element `/patient/name/firstname` which must have been defined with Search-Type set to "Standard" in the schema definition):

```
/patient[sex ~= 'male'] sortby (./name/firstname)
```

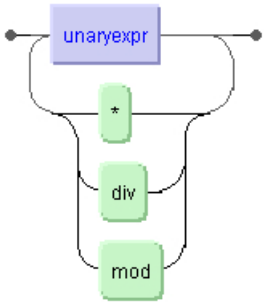
Related Expressions

AbsoluteLocationPath RelativeLocationPath

MultiplicativeExpr

Multiply or divide numeric values.

Syntax



MultiplicativeExpr

Description

A `MultiplicativeExpr` either consists of a simple `UnaryExpr` or of two or more operands to `UnaryExpr` separated by the operators that must be numeric values or can be converted to numeric values. If at least one operator is not a numeric value or cannot be converted, then the result is "NaN", a special value to indicate that this is not a number. Otherwise the numeric values are treated as double-length floating point numbers as specified in IEEE 754. This is also true for the division operator, although it is called `div` which is commonly used for the division of integer values (the typical division sign `/` is already used in path expressions). The modulo operation (using the operator `mod`) returns the remainder of a division.

In addition to "NaN" there are two other special values as specified in IEEE 754, namely positive and negative infinity. These are returned as "1.#INF" and "-1.#INF" respectively. You get these values if you perform queries such as `2 div 0` or `2 div -0`.

Compatibility

It corresponds to the expression `MultiplicativeExpr` defined in XPath, **Section 3.5, Rule 6**. This includes the XPath restriction that numbers cannot be used in scientific notation.

Examples

- Get the percentage of patients who died (divide the number of any descendant `deceased` nodes by the number of `patient` nodes and multiply that value by 100):

```
count(//deceased) div count(/patient) * 100
```

- Select all patients that were born in the first year of a decade:

```
/patient[born mod 10 = 0]
```

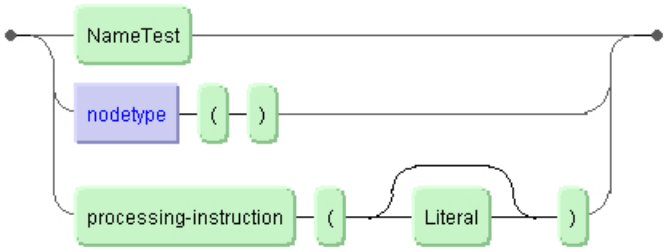
Related Expression

`AdditiveExpr`

NodeTest

Check if a node satisfies conditions on the type or name of a node.

Syntax



NodeTest

Description

A `NodeTest` defines type or name of a node in a `NodeTest`. It can be a `NameTest`, a node type as defined in `NodeType` or a processing instruction which optionally can have a name that conforms to the definition of a `Literal`.

A `NameTest` is either a name or a wildcard expression matching a set of names. If `NameTest` is "*", it matches any name, otherwise it matches a given name.

Using `NodeType()`, you can select nodes of the specified type ("comment", "node", or "text") on the relevant axis. As a further type of node, you can select processing instruction nodes (represented as `<?PITarget?>`) which may or may not have a `Literal`. A `Literal` is a sequence of characters that is enclosed either in double quotes (then it must not contain a double quote) or in single quotes (then it must not contain a single quote).

Note: Only processing instruction nodes (PIs) that are nested inside the root node element can be selected. Any PIs outside are not accounted for, such as the XML declaration that looks like a PI and declares a document to be an XML document (`<?xml version="1.0"?>`).

Compatibility

It corresponds to the expression `NodeTest` defined in XPath, **Section 2.3, Rule 7** with the exception that in X-Query it formally also subsumes the definition of an unnamed processing instruction which in XPath is included in the definition of `NodeType` instead. However, this difference is of no practical relevance.

`NameTest` and `Literal` correspond to the respective XPath expressions, as defined in the **Section 3.7**, Rules 37 and 29.

Examples

- Select all child elements of the context node (`NameTest` matching any name by using a wildcard; it is a shorthand for the relative location path `child::*`)

```
*
```

- Retrieve all children of the context node:

```
node()
```

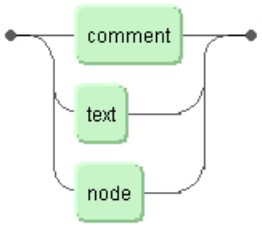
Related Expression

`NodeType Step`

NodeType

Restrict the type of a node in a `NodeTest`.

Syntax



NodeType

Description

A `NodeType` is used in a `NodeTest` and restrict the node-set to the specified type of node. See the description to `NodeTest` for more details.

Compatibility

It corresponds to the expression `NodeType` defined in XPath, **Section 3.7, Rule 38** with the exception that the XPath definition also includes a named processing instruction which in X-Query is subsumed in the definition of `NodeTest`. However, this difference is of no practical relevance.

Examples

- Select all comment nodes:

```
//comment()
```

- Retrieve all children of the context node:

```
node()
```

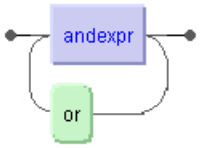
Related Expression

Step

OrExpr

Check if either of its Boolean conditions is true.

Syntax



OrExpr

Description

The operands are tested if either one of them evaluates to the Boolean value "true".

Compatibility

The operator name can only be specified using lower case.

It corresponds to the expression `OrExpr` defined in XPath, **Section 3.4, Rule 21**. Because of the X-Query definition of `AndExpr` you can also use `SequenceExpr`, `BetweenExpr`, and `ProximityExpr` as operands.

Example

Select all patients born in 1950 or 1960 (all `patient` nodes that satisfy the following predicate expression: there is an immediate child node `born` whose numeric value equals 1950 or there is an immediate child node `born` whose numeric value equals 1960):

```
/patient[born = 1950 or born = 1960]
```

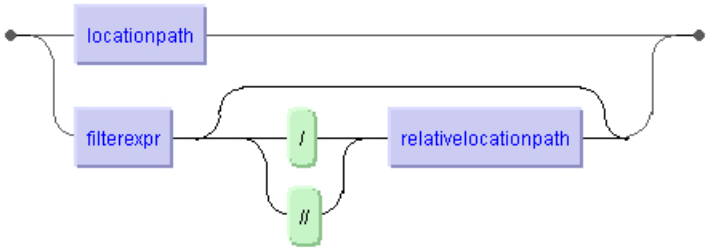
Related Expressions

AndExpr EqualityExpr

PathExpr

Select a set of nodes along a path.

Syntax



PathExpr

Description

Starting from a given point in a document tree, a `PathExpr` selects a node-set by following a location path. A path expression can be one of the following:

- a location path, selecting a set of nodes by following location steps, starting either at the document root (absolute location path) or at the context node (relative location path),
- a filter expression selecting a set of nodes e.g. as a result to a function call or an expression enclosed in parentheses, possibly followed by one or more predicates,
- a filter expression followed by the path operator `/`, followed by a relative location path which starts with the set of nodes resulting from the filter expression,
- a filter expression followed by the path operator `//`, followed by a relative location path which starts anywhere inside the set of nodes resulting from the filter expression

Compatibility

It corresponds to the expression `PathExpr` defined in XPath, **Section 3.3, Rule 19**.

Examples

- Select all patients (a relative `LocationPath` starting at the context node which is the doctype element node `patient` in our example):

```
patient
```

- Select the answer to all questions (a simple `FilterExpr` can be just a `PrimaryExpr` which is basically an expression without operator such as a `Number`):

```
42
```

- Get the number of all patients (a `FilterExpr` that identifies a node-set which results from calling the standard XPath function `count()` with all `patient` nodes as argument):

```
count(patient)
```

- Get the number of all patients who died (a `FilterExpr` that identifies a node-set which results from calling the standard XPath function `count()`; the argument is the set of all `patient` nodes that have a descendant node `deceased`):

```
count(patient[//deceased])
```

- Select the names of all doctors who have discharged patients or had to register a patient's death provided that the doctor's surname begins with "G" (a `FilterExpr` enclosed in a parenthesized `PrimaryExpr` followed by a `Predicate`: select the union of the set of any descendant `result/discharged` nodes and `result/deceased` nodes; from this node set select all descendant `name` nodes that have an immediate child element node `surname` whose value contains a word beginning with "G"):

```
((//result/discharged | //result/deceased)//name)[surname ~= 'G*']
```

Related Expressions

`Location Path`

`FilterExpr`

`SetExpr`

Predicate

Select a node subset by using a predicate expression

Syntax

Predicate

Description

You can use a `Predicate` in a location step or in a filter expression, both of which identify a node-set. This node-set is then restricted by applying the `PredicateExpr` can be used to filter a node set in a location step.

A predicate can either be a Boolean expression or a numeric expression. A numeric predicate `[n]` is short for `[position()=n]` which is true for all nodes that are the *n*th child element in the input node set. If the value of the predicate is not a number, then it is treated as a Boolean expression.

Compatibility

It corresponds to the expression `Predicate` defined in XPath, **Section 2.4, Rule 8**.

Examples

- Select all names in a patient's document (select all descendant child `name` nodes that are the first child element node of their respective parent nodes; in our example database, this returns six `name` nodes):

```
//name[1] ↩
```

- Select the names of all patients (select all `name` nodes that are the first child element node of any child of the root element; this restricts it effectively to `/patient/name[1]` which returns two `name` nodes):

```
/*/name[1]
```

- Select the names of all doctors (select all `name` nodes that are a child of any descendant `doctor` element node of any child of the root element and satisfy the following predicate expression: they are the last child element of their parent `doctor` element; since there is only one `name` node this is equivalent to `//doctor/name[position()=1]` so that three `name` nodes are returned):

```
//doctor/name[position()=last()]
```

- Select all doctors with a pager (all descendant `doctor` nodes that have a `pager` attribute node attached):

```
//name[@pager] ↩
```

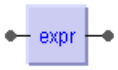
Related Expressions

`FilterExpr` `Step`

PredicateExpr

Any expression within a predicate.

Syntax



PredicateExpr

Description

A `PredicateExpr` can be any X-Query expression. If this expression results in a numeric value, then it is a numeric predicate, otherwise it is a Boolean predicate. See `Predicate` for more details.

Compatibility

It corresponds to the expression `PredicateExpr` defined in XPath, **Section 2.4, Rule 9**.

Example

Select all patients that have a next of kin (all `patient` nodes that have at least one immediate child element `nextofkin`):

```
/patient[nextofkin]
```

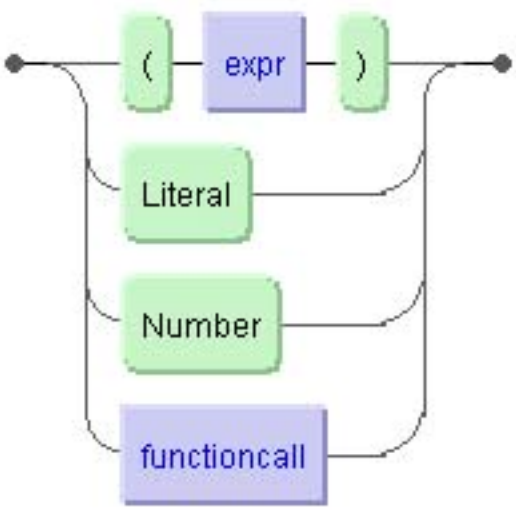
Related Expressions

`Expr` `Predicate` `Step`

PrimaryExpr

An expression without operators.

Syntax



PrimaryExpr

Description

A `PrimaryExpr` defines some basic expressions that are used in a `FilterExpr`. A `Literal` is a sequence of characters that is enclosed either in double quotes (then it must not contain a double quote) or in single quotes (then it must not contain a single quote).

Compatibility

It corresponds to the expression `PrimaryExpr` defined in XPath, **Section 3.1, Rule 15** with the restriction that XPath also supports a `VariableReference` which is not supported in X-Query.

`Literal` and `Number` correspond to the respective XPath expressions, as defined in the **Section 3.7**, Rules 29 and 30.

Examples

- The string value is a `Literal`:

```
'Atkins'
```

- Select all data of a patient except his or her name and any remarks (from all immediate child element nodes of the `patient` nodes select the siblings following the `name` node provided that they are not the last one):

```
(/patient/* after name)[position() != last()]
```

  You get this result only by enclosing the `SequenceExpr` in parentheses. Otherwise the predicate expression would be applied to the `name` node instead of to the whole sibling node set and this would yield an empty result set.

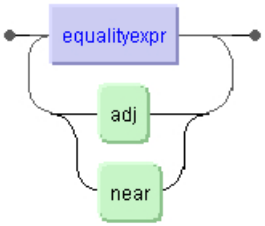
Related Expressions

OrExpr FunctionCall

ProximityExpr

Check if two values are equal or positioned close to each other.

Syntax



ProximityExpr

Description

A `ProximityExpr` can either be an `EqualityExpr` using the equality operators = and `!=` or the contains operator ~= or it can use the operators `adj` and `near`. The operator `adj` is used to locate adjacent values that are in the exact order as specified with the `adj` operator. The operator `near` is used to locate adjacent values irrespective of exact order. The `adj` and `near` operators can only be used on the right side of an expression with the contains operator.

Compatibility

There is no `ProximityExpr` in XPath.

Examples

- Select all `patient` nodes that have the word value "professional" followed immediately by the word "diver" for the element `occupation`:

```
/patient[occupation ~= 'professional' adj 'diver']
```

Note: This is equivalent with the expression `[occupation ~= 'professional diver']`.

- Select all `patient` nodes that have the word "professional" node followed immediately by the word "diver" (or vice versa) for the element `occupation`:

```
/patient[occupation ~= 'professional' near 'diver'] ↩
```

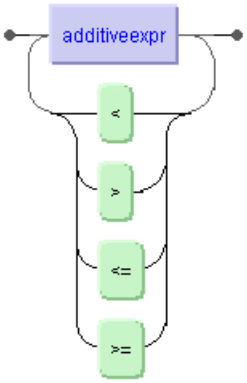
Related Expression

`BetweenExpr`

RelationalExpr

Compare two values by using relational operators.

Syntax



RelationalExpr

Description

A `RelationalExpr` can be a simple `AdditiveExpr` or two or more `AdditiveExpr` expressions that are operands to one of the four relational operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal). The comparison results in a Boolean value "true" or "false".

You can compare numeric values as well as string values, which are compared according to lexicographic ordering. If there is a collation defined for either of the operands or if both operands have the same collation defined, then the comparison is based on this collation, otherwise it is character-based.

Compatibility

It is compatible to the expression `RelationalExpr` defined in XPath, **Section 3.4, Rule 24**. However, in XPath `RelationalExpr` is restricted to comparison of numeric values.

In contrast to XSLT there is no need to use entity references for the "<" and ">", since the query string is not processed as an XML instance. But in XSLT a `RelationalExpr` appears as an attribute

value and must therefore be referenced; the first example would look as follows: `/patient/[born &lt; 1959]`.

Examples

- Select the addresses of all patients whose surname is ordered before "Bl". The patient "Bloggs" is not in the result set, since "Bl" is positioned before "Bloggs" in the lexicographical order.

```
/patient/address[../name/surname < 'Bl']
```

- Check if the mortality rate is below 10% (check if the number of `patient` nodes multiplied by 10 is greater than the number of `deceased` nodes)

```
count(/patient)*10 > count(//deceased)
```

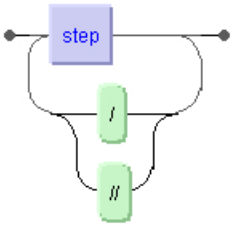
Related Expressions

BetweenExpr EqualityExpr

RelativeLocationPath

Select nodes on a relative location path starting at the context node.

Syntax



RelativeLocationPath

Description

A `RelativeLocationPath` consists of a sequence of one or more location steps (`Step`) separated by one of the path operators `/` or `//`. The operator `/` selects the child nodes of the context node, while `//` selects all descendant nodes of the context node, since it is an abbreviation for the full XPath expression `descendant-or-self::node()`.

Compatibility

It corresponds to the expression `RelativeLocationPath` defined in XPath, **Section 2.4, Rule 3**.

Examples

- Select all patients (a `RelativeLocationPath` consisting of a single `AbbreviatedStep`: it selects the child nodes of the context node, which are the `patient` element nodes)

```
patient
```

- Select all doctors who discharged a patient (a `RelativeLocationPath` that selects every `doctor` element node which is a child of a `discharged` element node which is a child of a `result` element node which is a child of a `patient` element node which is a child of the context node):

```
patient/result/discharged/doctor
```

- Select all therapies (an abbreviated `RelativeLocationPath` that selects every `medication` element node which is a child of a `therapy` element node which is a descendant of the context node):

```
.//therapy/medication
```

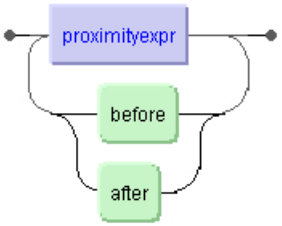
Related Expressions

`Step`

SequenceExpr

Select nodes based on position.

Syntax



SequenceExpr

Description

A `SequenceExpr` consists either of a single `ProximityExpr` or of two or more proximity expressions that are separated by the operators `before` or `after`. With these operators you can select nodes based on sibling positioning.

The left operand of the `before` or `after` operator must be a set of sibling nodes and the right-hand operand must be an element of this node-set (otherwise an empty node-set will be returned as a result).

Compatibility

There is no `SequenceExpr` in XPath. You can use the operators `before` and `after` to simulate the XPath way of selecting nodes along one of the axes `preceding-sibling` or `following-sibling`, see the second and third example.

Examples

- Select remarks to a patient (all descendant `remarks` element nodes are sibling to some element node `therapy` and positioned after that node; in our example database this is true for only one node; see also next example):

```
//remarks after therapy
```

- Select street and housenumber of all patients' addresses (select all element nodes that are child nodes of `patient/address` and select only those that in the document order of siblings are positioned before the `city` element):

```
/patient/address/* before city
```

- Select all data of a patient except his or her name and any remarks (from all immediate child element nodes of the `patient` nodes select the siblings following the `name` node provided that they are not the last one):

```
(/patient/* after name)[position() != last()]
```

  Please note that you have to use the parentheses here. Otherwise the predicate expression would be applied to the `name` node and this would yield an empty result set.

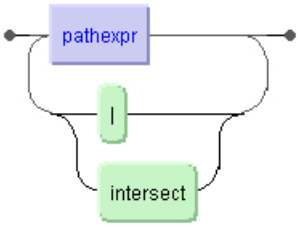
Related Expressions

`ProximityExpr`

SetExpr

Select the union or intersection of node sets.

Syntax



SetExpr

Description

With a `SetExpr` you can select the union or intersection of node sets. A union set is constructed by using the operator `|` and contains all nodes that are in any one of the node sets. An intersection is constructed by using the operator `intersect` and contains only those nodes that are in all nodes.

Compatibility

It corresponds to the expression `UnionExpr` defined in XPath, **Section 3.3, Rule 18**. However, X-Query adds the `intersect` operator which is not present in XPath.

Examples

- Select the union of all `medication` descendants and `diagnosis` descendants of patients:

```
/patient//medication | /patient//diagnosis
```

- Select the intersection of all patients born in 1950 and patients with medication in tablet form among all male patients:

```
/patient[born=1950] intersect /patient[//medication/type[@form='tablet']] intersect ↩
/patient[sex ~= 'male']
```

- Certainly, you can write this query easier, for example:

```
/patient[born=1950 and .//medication/type[@form='tablet'] and sex ~= 'male']
```

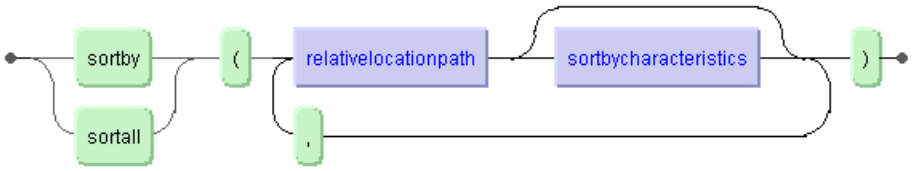
Related Expressions

Location Path PrimaryExpr Predicate

SortByClause

Sort a set of nodes within or across documents in ascending or descending order. The sort criterion is specified by a relative location path.

Syntax



SortByClause

Description

In Tamino, you can use a `SortByClause` for sort operations using the following keywords:

`sortby`
: nodes within a document (intra-document sort)

`sortall`
: multiple documents (inter-document sort)

If there is a collation defined for the element or attribute that is used as sort criterion, then the sort will be based on that collation. You can specify multiple sort criteria by separating them with a comma. The default sorting order is ascending, but you can also specify a descending order (see `SortByCharacteristics`).

Usage Information

- It is an error, if the sort argument expression evaluates to more than one node. For example, given the XML document

```
<A>
  <B>Item1</B>
  <B>Item2</B>
</A>
```

the error INOXIE8309 will occur when trying `A sortby (B)` or `A sortall (B)`, since it is not clear which `B` node to take as the sort criterion. In cases like these, the following table helps you finding a unique sort expression (*sort* standing for `sortby` and for `sortall`):

`A sort (B[1])`	first element
`A sort (B[last()])`	last element
`A sort (B[. = max(../B)])`	maximum numerical value
`A sort (B[. = min(../B)])`	minimum numerical value
`A sort (B[not(. < ../B)])`	maximum string value
`A sort (B[not(. > ../B)])`	minimum string value

- To guarantee a type-conforming sort, you should use a structure index for the doctype in question that uses at least the setting "CONDENSED" .
- It is an error to address the parent nodes after applying a sort operation. For example:

```
/patient/name sortall (.) /..
```

will yield error INOXIE8345.

- You can use only one `sortall` operation at a time. For example,

```
(/patient/name sortall (.)) sortall (.)
```

will yield error INOXIE8355.

The following restriction applies to `sortby` operations within a document:

- You cannot sort by the results of a function call such as `/*/* sortby(position() desc)`.

Compatibility

There is no `SortByClause` in XPath.

In previous versions, a `sortby` expression that is immediately located after the root node (e.g. `/A sortby (.)`, has the same semantics as `/A sortall (.)` now, namely performing an inter-document sorting. This has been retained for compatibility.

Examples

- Select all `firstname` descendant nodes and return them sorted within each document:

```
//firstname sortby (.)
```

This yields the following list of result nodes:

```
<firstname ino:id="1">Dorothy</firstname>
<firstname ino:id="1">John</firstname>
<firstname ino:id="1">John</firstname>
<firstname ino:id="1">Paul</firstname>
<firstname ino:id="2">A.</firstname>
<firstname ino:id="2">Fred</firstname>
```

- Select all firstname nodes in ascending order across documents

```
//firstname sortall (.)
```

An error INOXIE8355 will occur, because the example data uses `firstname` nodes at several positions in the document tree. You can work around this by using a union set of all six possible nodes explicitly:

```
(/patient/name/firstname | /patient/nextofkin/name/firstname | ↩
/patient/submitted/doctor/name/firstname |
/patient/result/deceased/doctor/name/firstname | ↩
/patient/result/discharged/doctor/name/firstname |
/patient/result/transferred/doctor/name/firstname) sortall (.)
```

This yields the correct list, namely all occurrences of `firstname` nodes sorted across all document instances:

```
<firstname ino:id="2">A.</firstname>
<firstname ino:id="1">Dorothy</firstname>
<firstname ino:id="2">Fred</firstname>
<firstname ino:id="1">John</firstname>
<firstname ino:id="1">John</firstname>
<firstname ino:id="1">Paul</firstname>
```

- Get an alphabetically sorted list of all brands of medicine which has been used in the form of tablets and in liquid form.

```
(//type[@form='tablet'] | //type[@form='liquid']) sortall (@brand)
```

- Select all male patients sorted by their surnames using an inter-document sort; sort the node-set resulting from `/patient[sex ~= 'male']` by the element `/patient/name` and return the list in descending order:

```
/patient[sex ~= 'male'] sortall (name desc)
```

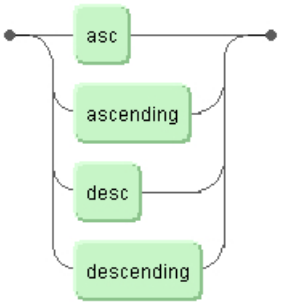
Related Expressions

Step SetExpr, FilterExpr

SortByCharacteristics

Define the order in which objects are to be sorted.

Syntax



SortByCharacteristics

Description

In Tamino, you can use `SortByClause` for sorting database objects. The `SortByCharacteristics` defines the order in which database objects should be sorted, which can be ascending (using `ascending` or `asc` as shorthand) or descending (using `descending` or `desc` as shorthand). The default sorting order is ascending. See `SortByClause` for a more detailed description of how sorting works.

Compatibility

There is no `SortByCharacteristics` in XPath.

Example

Select all patients sorted by their submission date beginning with the most current one; patients who were submitted on the same day, should be sorted alphabetically by their name:

```
/patient sortall (submitted/date desc, name asc)
```

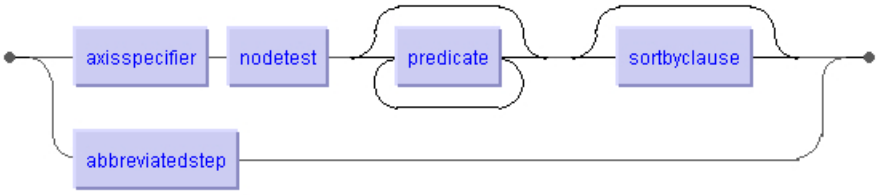
Related Expressions

SortByClause

Step

Select a set of nodes starting from a given node-set.

Syntax



Step

Description

A step is specified in a location path in order to select a set of nodes relative to a context node. A step can also be specified using an abbreviated syntax.

Compatibility

It corresponds to the expression `Step` defined in XPath, **Section 2.1, Rule 4**.

Examples

- Select all `medication` nodes that are children of a `therapy` node that are children of a `patient` node:

```
/patient/therapy/medication
```

- Using abbreviated syntax, select all `diagnosis` elements of the current context node that have a value of "DCI":

```
//diagnosis[.='DCI']
```

- Using abbreviated syntax, retrieve all `diagnosis` elements whose parent node has the element `symptoms` containing the value "headache":

```
//diagnosis[../symptoms ~= 'headache']
```

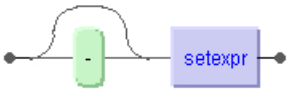
Related Expressions

AxisSpecifier NodeTest Predicate

UnaryExpr

Change the sign of a number.

Syntax



UnaryExpr

Description

The `UnaryExpr` is a unary operator which changes the sign of a `Number`. It takes as argument a `SetExpr` which may also be a node set. In that case, it tries to convert the value of the respective node of the last document in the result set into a numeric value and changes the sign of that number

Compatibility

It corresponds to the expression `UnaryExpr` defined in XPath, **Section 2.4, Rule 3**.

Examples

- This is simply a `Number`, the numeric value -3.

```
-3
```

- Change the sign of the numeric value -3.

```
-(-3)
```

- Subtract the year of birth of the last patient from the numeric value -2001 (no application of `UnaryExpr`):

```
-2001-patient/born
```

- Subtract the year of birth of the last patient from 2001 and change it in a negative value:

```
-(2001-patient/born)
```

Related Expressions

`AdditiveExpr`

3 Functions

This chapter describes the functions available in X-Query. Many of them are defined in XPath with the same syntax and semantics. Please refer to the **XPath specification** for a more detailed description. Functions that are not present in XPath are marked with an asterisk. Some of the functions that are present in XPath are not implemented in X-Query, but you can extend the functionality of X-Query by writing server extensions implementing the desired function. These extensions can then be registered within the database server so that they are available for all further queries.

Category	Function	Short Description
Type conversion	`boolean()`	convert argument to a Boolean value
	`number()`	convert argument to a number
	`string()`	convert argument to a string
Arithmetic functions	`ceiling()`	return the smallest integer that is greater than or equal to the argument's numeric value
	`floor()`	return the largest integer that is less than or equal to the argument's numeric value
	`round()`	return the integer that is closest to the argument's numeric value
String functions	`starts-with()`	test whether one string starts with another string
Aggregation	`avg()` *	return the average value of a set of numeric values that the argument's node set contain
	`count()`	return the number of nodes present in the argument's node set
	`min()` *	return the smallest value of a set of numeric values that the argument's node set contain

Category	Function	Short Description
	`max()` *	return the largest value of a set of numeric values that the argument's node set contain
	`sum()`	calculate the total of a set of numeric values that the argument's node set contain
Node Names	`name()`	return a qualified name representing the name of a node
Boolean functions	`false()`	return the Boolean value "false"
	`not()`	return the negated value of its argument
	`true()`	return the Boolean value "true"
Context information	`last()`	return the position number of the last node in the list
	`position()`	return the position number of the context node
Analysis	`ino:explain()` *	return query execution plan

avg

Return the average of the argument's numeric value.

Syntax

```
avg (nodeset])
```

Description

This function returns the average of the numeric value of the nodes of a node set. If you specify a node set that contains more than one node, all nodes of the set are taken for evaluation.

If necessary, node values are converted to numerical values. If a node value cannot be converted, then NaN is used. This function returns NaN, if at least one argument is NaN.

Compatibility

There is no `avg` function in XPath.

Example

Compute the average age of all male patients (subtract from the current year the average value of the aggregated `/patient/born` nodes):

```
2002 - avg(/patient/born[../sex~='male'])
```

min

Return smallest value of the argument's numeric value.

Syntax

```
min (nodeset[, nodeset])
```

Description

This function returns the minimum numeric value of the nodes of one or more node sets. If you specify a node set that contains more than one node, all nodes of the set are taken for evaluation.

If necessary, node values are converted to numerical values. If a node value cannot be converted, then NaN is used. This function returns NaN, if at least one argument is NaN.

Compatibility

There is no `min` function in XPath.

Example

Retrieve the year of birth of the oldest patient (aggregate all `/patient/born` nodes and determine the minimum numerical value):

```
min(/patient/born)
```

max

Return largest value of the argument's numeric value.

Syntax

```
max (nodeset[, nodeset])
```

Description

This function returns the maximum numeric value of the nodes of one or more node sets. If you specify a node set that contains more than one node, all nodes of the set are taken for evaluation.

If necessary, node values are converted to numerical values. If a node value cannot be converted, then NaN is used. This function returns NaN, if at least one argument is NaN.

Compatibility

There is no `max` function in XPath.

Example

Retrieve the year of birth of the youngest patient (aggregate all `/patient/born` nodes and determine the maximum numerical value):

```
max(/patient/born)
```

ino:explain

Retrieve information about query execution for analysis and optimization.

Syntax

```
ino:explain (query[, level])
```

Description

This function provides information about the execution plan of a given query. As it is a Tamino-internal function, it has the namespace prefix `ino`. It takes as argument any valid query expression (`OrExpr`) and an optional *`level`* of explanation, which can be one of the values "path" and "tree". If *`level`* is omitted, then basic information about the processing steps involved is provided. It returns information about the execution plan of the query inside the regular `<xql:result>` node of the standard `<ino:response>`. This information is wrapped up in a new element `<ino:explanation>`.

The execution time of a query depends on the number and kind of processes that are needed to resolve the query. A query is processed in Tamino as follows:

1. **Query Parser**
   It takes as input the query string and parses it. If it does not conform to the syntax rules of X-Query then an error message will be returned indicating the type of error. If the query can be successfully parsed, it delivers an input tree for the optimizer.
2. **Query Optimizer**
   The optimizer tries to optimize queries on the level of X-Query by applying a number of transformations and using the information from the corresponding schema. It performs amongst others the following kinds of transformation as far as they are applicable:
   - descendants expansion: abbreviated relative or absolute location paths are expanded into a disjunction of unabbreviated paths
   - wildcard expansion: if you use `*` in a `NameTest`, then it will be expanded into a disjunction of all matched nodes. (e.g. `insurance/*` would be expanded into `insurance/company or insurance/policynumber)`
   - path evaluation in predicate expressions: reformulate path references such as `a/b/c[../d]` into `a/b[d]/c`
   - `not()` replacement: applying de Morgan rules, the optimizer tries to replace expressions that use a call of the `not()` function (e.g. an expression such as `not(surname = "Atkins")` would be transformed into `surname != "Atkins"`

Example: You use an abbreviated location path in your query such as in `patient[.//surname ~= 'Atkins']`. From the schema, the optimizer detects that the element `surname` can occur at six different positions in a document tree: `patient/name/surname`, `patient/nextofkin/name/surname`, and `doctor/name/surname`, where `doctor` can appear under `patient/submitted`, `result/discharged`, `result/transferred`, and `result/deceased`. The optimizer then transforms the filter expression into the following disjunction:

```
patient[./name/surname ~= 'Atkins' or
        ./nextofkin/name/surname ~= 'Atkins' or
        ./submitted/doctor/name/surname ~= 'Atkins' or
        ./result/discharged/doctor/name/surname ~= 'Atkins' or
        ./result/transferred/doctor/name/surname ~= 'Atkins' or
        ./result/deceased/doctor/name/surname ~= 'Atkins']
```

Instead of searching the complete document tree only these nodes must be visited to see if the predicate expression holds. As a result, the optimizer delivers a modified tree.

3. **Processor-specific Optimizer**
   This component optimizes the tree with regard to the special needs of the next processing components. For both, the index processor and the postprocessor, a tree will be generated that best suits their needs.
4. **Index Processor**
   This component evaluates all predicates containing indexed element nodes. It is further responsible for accessing documents and schemas from the database as well as for composing the XML document that contains the query result. It is possible that the index processor creates a superset of the query result which then has to be restricted in the next step. If no further processing is necessary, then the index processor returns the result set as an instance of `xql:result`.
5. **Postprocessor**
   Since typically not every node is indexed, there is another processing stage that evaluates expression with non-indexed nodes. The postprocessor also restricts the result set if the index processor generated a superset by scanning a doctype or collection. Furthermore it also makes calls to any query functions. If invoked it will return the complete query result.

A call to `ino:explain` provides information about which processing components are involved, to what degree the query can be optimized, and the work load of the index processor and the postprocessor. According to the selected explanation level a different amount of information is returned inside an element called `ino:explanation`. This element uses two attributes, `ino:document_processing` and `ino:preselection` that are used as flags and indicate the way the query is processed. Inside `ino:explanation` a set of elements can appear that share the namespace prefix `xop`. They correspond to expressions in X-Query. For example, the element `xop:matches` represents the match operator '~=', and the node `<xop:literal xop:value="Atkins" />` represents the literal string constant "Atkins". The sections below describing the explanation levels contain more information about which principal elements and attributes of the `xop` namespace are important and how they can be used for the purpose of query analysis and optimization.

Note: The information that is returned by a call of `ino:explain()` shows the internal structure of query processing and is subject to change without prior notice if this is necessary because of improvements in the underlying mechanism.

No Explanation Level

In the query result only `ino:explanation` appears along with its two required attributes. They mean:

- `ino:preselection`: indicates whether a full scan of the doctype or collection will be performed for the given query. If "TRUE" there is some restriction in the query that can be processed by the index processor, which means that there may be documents that can be rejected without calling the postprocessor. "FALSE" indicates a full scan of the doctype or collection.
- `ino:postprocessing`: if "TRUE" then the postprocessor will be called.

To retrieve the execution plan for a query looking for patient whose surname contains "Atkins":

```
ino:explain(patient[.//surname ~= "Atkins"])
```

The result from the server looks like this (only showing the relevant `<xql:result>` node):

```
<xql:result>
  <ino:explanation ino:preselection="TRUE" ino:postprocessing="TRUE" />
</xql:result>
```

Explanation Level "path"

This level shows the query after the optimizer run. Each step of a location path is represented by its own `xop:path` element. Nesting of `xop:path` elements means traversing the location path one step further along the child axis. Any instance of `xop:path` uses these attributes:

- `xop:name`: name of the element, always present
- `xop:searchtype`: search type of the element as defined in the schema, only present if there are no child elements
- `xop:maptype`: mapping type of the element as defined in the schema, if none is defined, the value is "no".

Using the example from above the returned `ino:explanation` node contains the following series of `xop:path` elements (the larger middle part deleted for brevity):

```
<xop:path xop:name="patient" xop:maptype="native">
  <xop:path xop:name="name" xop:maptype="infofield">
    <xop:path xop:name="surname" xop:maptype="infofield" />
  </xop:path>
  <xop:path xop:name="nextofkin" xop:maptype="infofield">
    <xop:path xop:name="name" xop:maptype="infofield">
      <xop:path xop:name="surname" xop:maptype="infofield" />
    </xop:path>
  </xop:path>
  ...
  <xop:path xop:name="address" xop:maptype="infofield" />
</xop:path>
```

The "patient" element contains the element name which in turn contains the element surname, each of them with their schema mapping type definition in the `xop:maptype` attribute.

Explanation Level "tree"

The structure of `ino:explanation` acknowledges the query trees that are built and modified during query processing. For each tree that is used during processing, there is a corresponding `xop:querytree` element that are distinguished by the attribute `xop:treetype` as follows.

- **Input Tree for Optimizer**
  This tree represents the original query and is always included in the output of `ino:explain()`. For example the query `patient[.//surname ~= "Atkins"]` is represented as follows (nested elements are indented for better readability):

```
<xop:querytree xop:treetype="input tree for optimization">
  <xop:list_context xop:collection="Patient" />
  <xop:element_children />
  <xop:nametest xop:name="patient" />
  <xop:filter>
    <xop:matches>
      <xop:transparent>
        <xop:curcontext />
        <xop:descendant_elem />
        <xop:nametest xop:name="surname" />
      </xop:transparent>
      <xop:literal xop:value="Atkins" />
    </xop:matches>
  </xop:filter>
  <xop:element_children />
  <xop:nametest xop:name="address" />
</xop:querytree>
```

  From the collection "Patient" (`<xop:list_context xop:collection="Patient" />`), those child element nodes whose name equals "patient" (`<xop:nametest xop:name="patient" />`) are selected that satisfy the condition set in the filter expression (`xop:filter`). The filter contains an expression with the match operator (`xop:matches`) with two operands appearing in document order. The left operand is enclosed in `xop:transparent` as a sequence of elements that selects

starting from the context node <xop:curcontext /> any descendant elements (<xop:descendant_elem />) whose name equals "surname" (<xop:nametest xop:name="surname" />). If the value of these descendant "surname" element nodes matches the literal value "Atkins" (<xop:literal xop:value="Atkins" />), then they satisfy the filter expression and form the node set from which all child element nodes with the name "address" should be selected as result of the query.

- **Output Tree from Optimizer**
  This tree is not returned when the optimizer run yields an empty result. Using our previous example, the optimizer converts the original query into a query containing a disjunction as outlined above. An excerpt of the resulting query tree is shown below. Only the first two clauses of the disjunction (xop:or) are complete; the structure of the other four clauses is "folded" and indicated by an ellipsis:

```
<xop:querytree xop:treetype="output tree from optimization">
  <xop:list_context xop:collection="Patient" />
  <xop:element_children />
  <xop:nametest xop:name="patient" xop:maptype="native" xop:key="id0000000181" />
  <xop:filter>
    <xop:or xop:preselectable="FALSE">
      <xop:matches xop:preselectable="FALSE">
        <xop:transparent>
          <xop:nametest xop:name="name" xop:maptype="infofield" ↩
xop:key="id0000000183" />
          <xop:element_children />
          <xop:nametest xop:name="surname" xop:maptype="infofield" ↩
xop:key="id0000000184" />
        </xop:transparent>
        <xop:literal xop:value="Atkins" />
      </xop:matches>
      <xop:matches xop:preselectable="FALSE">
        <xop:transparent>
          <xop:nametest xop:name="nextofkin" xop:maptype="infofield" ↩
xop:key="id0000000201" />
          <xop:element_children />
          <xop:nametest xop:name="name" xop:maptype="infofield" ↩
xop:key="id0000000203" />
          <xop:element_children />
          <xop:nametest xop:name="surname" xop:maptype="infofield" ↩
xop:key="id0000000204" />
        </xop:transparent>
        <xop:literal xop:value="Atkins" />
      </xop:matches>
      <xop:matches xop:preselectable="FALSE"> ...
      <xop:matches xop:preselectable="FALSE"> ...
      <xop:matches xop:preselectable="FALSE"> ...
      <xop:matches xop:preselectable="FALSE"> ...
    </xop:or>
  </xop:filter>
  <xop:element_children />
```

```
  <xop:nametest xop:name="address" xop:maptype="infofield" xop:key="id0000000190" />
</xop:querytree>
```

In addition to the transformation of the query, the following attributes have been added:

- `xop:key` has as value the internal ID assigned to this node.
- `xop:maptype` holds schema, possible values are "infofield", "native".
- `xop:preselectable` indicates with the Boolean values "TRUE" and "FALSE" if this part of the query can be processed by the index processor.

- **Output Tree for Index Processor**
  This is the tree produced by the process-specific optimizer to be used by the index processor. It is not returned if database access is not necessary.

```
<xop:querytree xop:treetype="output tree for index processor">
  <xop:list_context xop:collection="Patient" />
  <xop:element_children />
  <xop:nametest xop:name="patient" xop:maptype="native" xop:key="id0000000181" />
  <xop:element_children />
  <xop:nametest xop:name="address" xop:maptype="infofield" xop:key="id0000000190" />
</xop:querytree>
```

- **Output Tree for Postprocessor**
  This is the tree produced by the process-specific optimizer to be used for the post processor. It is not returned if postprocessing is not necessary.

```
<xop:querytree xop:treetype="output tree for document processor">
  <xop:list_context xop:collection="Patient" />
  <xop:element_children />
  <xop:nametest xop:name="patient" />
  <xop:filter>
    <xop:matches>
      <xop:transparent>
        <xop:curcontext />
        <xop:descendant_elem />
        <xop:nametest xop:name="surname" />
      </xop:transparent>
      <xop:literal xop:value="Atkins" />
    </xop:matches>
  </xop:filter>
  <xop:element_children />
  <xop:nametest xop:name="address" />
</xop:querytree>
```

Compatibility

Neither in XPath nor in XSLT is there an equivalent for this Tamino-specific function.

Example

Retrieve information about the execution plan of a query looking for patients whose surnames contain "Atkins":

```
ino:explain(patient[.//surname ~= "Atkins"])
```

The result from the server looks as follows (only showing the relevant `<xql:result>` node):

```
<xql:result>
  <ino:explanation ino:preselection="FALSE" ino:postprocessing="TRUE"/>
</xql:result>
```

So there is no index on the `surname` element node and a postprocessor run is necessary. You can do the following to minimize processing costs:

- If you know the schema, rewrite the abbreviated path:

  ```
  patient[name/surname ~= 'Atkins']
  ```

- If you know that the string value you're looking for is the complete value, then use a standard quality operator such as:

  ```
  patient[name/surname = 'Atkins']
  ```

- Define an index onto `surname`.

Index