XPath v2.0
Quick Reference

ver 1/0

xmlhelp@dvint.com
http://www.xml.dvint.com

1 Namespaces

`http://www.w3.org/2001/XMLSchema`, prefixed as `xs`.
`http://www.w3.org/2005/xqt-errors` prefixed as `err`
`http://www.w3.org/2005/xpath-functions` prefixed as `fn`

2 Document Order §2.4.1

Document order is the order in which nodes appear in the XML serialization of a document. Document order is **stable**, which means that the relative order of two nodes will not change during the processing of a given expression, even if this order is implementation-dependent. The node ordering that is the reverse of document order is called **reverse document order**.

3 Atomization §2.4.2

Atomization is applied to a value when the value is used in a context in which a sequence of atomic values is required. The result of atomization is either a sequence of atomic values or a type error. **Atomization** of a sequence is defined as the result of invoking the `fn:data` function on the sequence.

Atomization is used in processing the following types of expressions:

- Arithmetic expressions
- Comparison expressions
- Function calls and returns
- Cast expressions

4 Effective Boolean Value §2.4.3

The **effective boolean value** of a value is defined as the result of applying the `fn:boolean` function to the value. The effective boolean value of a sequence is computed implicitly during processing of the following types of expressions:

- Logical expressions (`and`, `or`)
- The `fn:not` function
- Certain types of
  - predicates, such as `a[b]`
  - Conditional expressions (`if`)
  - Quantified expressions (`some`, `every`)
  - General comparisons, in XPath 1.0 compatibility mode.

5 Types §2.5

A **sequence type** is a type that can be expressed using the SequenceType syntax. Sequence types are used whenever it is necessary to refer to a type in an XPath expression.

A **schema type** is a type that is or could be defined using the facilities of XML Schema. Every schema type is either a **complex type** or a **simple type**; simple types are further subdivided into **list types**, **union types**, and **atomic types**.

Atomic types represent the intersection between the categories of sequence type and schema type. An atomic type, such as `xs:integer` or `my:hatsize`, is both a sequence type and a schema type.

Predefined Schema Types §2.5.1

- `xs:untyped` is used for an element node that has not been validated, or has been validated in `skip` mode.
- `xs:untypedAtomic` is an atomic type that is used to denote untyped atomic data, such as text that has not been assigned a more specific type.
- `xs:dayTimeDuration` is derived by restriction from `xs:duration` restricted to contain only day, hour, minute, and second components.
- `xs:yearMonthDuration` is derived by restriction from `xs:duration` restricted to contain only year and month components.
- `xs:anyAtomicType` is an atomic type that includes all atomic values (and no values that are not atomic). Its base type is `xs:anySimpleType` from which all simple types, including atomic, list, and union types, and primitive atomic types, such as `xs:integer`, `xs:string`.

Sequence Types §2.5.3

`empty-sequence()` or ItemType Occurrence_Indicator
ItemType = KindTest or `item()` or AtomicType
AtomicType = QName
An Occurrence Indicator specifies the number of items in a sequence, as follows:

- `?` matches zero or one items
- `*` matches zero or more items
- `+` matches one or more items
- none matches one item only and is required

As a consequence of the following rules, any sequence type whose occurrence indicator is `*` or `?` matches a value that is an empty sequence.

- `empty-sequence()` matches a value that is the empty sequence.
- An itemType with an occurrence indicator matches a value if the number of items in the value matches the occurrence indicator and the ItemType matches each of the items in the value.

6 Comments §2.6

Comments are strings, delimited by the symbols `(:` and `:)`. Comments are lexical constructs only, and do not affect expression processing. Comments may be nested and used anywhere ignorable whitespace is allowed .

7 Primary Expressions §3.1

Literals §3.1.1

Integer = 123 Decimals = 1.23 Doubles = 1.23 e+2
“String” or ‘string’ Escape Quote = '""' Escape Apos = "''"

Variable References §3.1.2

`$QName` Two variable references are equivalent if their local names are the same and their namespace prefixes are bound to the same namespace URI in the statically known namespaces. An unprefixed variable reference is in no namespace.

Parenthesized Expressions §3.1.3

empty sequence = ()

Parentheses enforce a particular evaluation order in expressions that contain multiple operators.

Context Item Expression §3.1.4

- A **context item expression** evaluates to the context item, which may be either a node (as in the expression `fn:doc("bib.xml")/books/book[fn:count(./author)>1]`) or an atomic value (as in the expression `(1 to 100)[. mod 5 eq 0]`).
- The **context item** is the item currently being processed. An item is either an atomic value or a node.When the context item is a node, it can also be referred to as the context node. The context item is returned by an expression consisting of a single dot (`.`).
- If the context item is undefined, a context item expression raises a dynamic error

3.1.5 Function Calls

- A function call consists of a QName followed by a parenthesized list of zero or more expressions, called arguments. If the QName in the function call has no namespace prefix, it is considered to be in the default function namespace.
- If the expanded QName and number of arguments in a function call do not match the name and arity of a function signature in the static context, a static error is raised.

8 Path Expressions §3.2

A series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//".

Steps §3.2.1

Axis specifier, node test, zero or more predicates

Axes §3.2.1.1

- **Forward**
  - `child::` `descendant::` `descendant-or-self::` `self::` `following::` `following-sibling::`
- **Reverse**
  - `ancestor::` `ancestor-or-self::` `parent::` `preceding::` `preceding-sibling::`
- **Other**
  - `namespace::` `attribute::`

Predicates §3.2.2

- `[expr]`

Abbreviated Syntax §3.2.4

- (nothing) = `child::`
- `@` = `attribute::`
- `//` = /descendant-or-self::node()/
- `.` = `self::node()`
- `..` = `parent::node()`
- `/` = Node tree root

Node/Kind Tests §2.5.4.1, 3.2.1.2

- `name`
- `prefix:name`
- `*`
- `prefix:*`
- `attribute()` `attribute(*)` `attribute(*, TypeName)` `attribute(AttributeName)` `attribute(AttributeName, TypeName)`
- `comment()`
- `document-node()` `document-node(element(book))`
- `element()` `element(*)` `element(*, TypeName ?)` `element(*, TypeName)` `element(ElementName)` `element(ElementName, TypeName ?)` `element(ElementName, TypeName)`
- `item()`
- `node()`
- `processing-instruction()` `processing-instruction(N)`
- `schema-attribute(AttributeName)` `schema-element(ElementName)`
- `text()`

9 Sequence Expressions §3.3

Sequences are never nested--for example, combining the values 1, (2, 3), and ( ) into a single sequence results in the sequence (1, 2, 3).

Constructing Sequences §3.3.1

The **comma operator**, evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence.

- `10, 1, 2, 3, 4)` a sequence of five integers:
- `(10, (1, 2), (), (3, 4))` four sequences evaluates to `10, 1, 2, 3, 4.`

A **range expression** result is a sequence containing the two integer operands and every integer between the two operands, in increasing order.

- `(10, 1 to 4)` evaluates to the sequence `10, 1, 2, 3, 4.`
- `15 to 10` a sequence of length zero.
- `fn:reverse(10 to 15)` evaluates to the sequence `15, 14, 13, 12, 11, 10.`

Filter Expressions §3.3.2

- `$products[price gt 100]` = return only those products whose price is greater than 100
- `(1 to 100)[. mod 5 eq 0]` the integers from 1 to 100 that are divisible by 5
- `(21 to 29)[5]` result is the integer 25
- `$orders[fn:position() = (5 to 9)]` returns the fifth through ninth items in the sequence bound to variable `$orders`
- `$book/(chapter | appendix)[fn:last()]` returns the last chapter or appendix within the book bound to variable `$book`
- `fn:doc("zoo.xml")/fn:id('tiger')` returns the element node within the specified document whose ID value is `tiger`

Combining Node Sequences §3.3.3

`union | intersect except`

All these operators eliminate duplicate nodes from their result sequences based on node identity. The resulting sequence is returned in document order. If an operand contains an item that is not a node, a type error is raised.

- `$seq1` is bound to (A, B) `$seq2` is bound to (A, B) `$seq3` is bound to (B, C)
- `$seq1 union $seq2` evaluates to the sequence (A, B)
- `$seq1 intersect $seq2` evaluates to the sequence (A, B)
- `$seq2 except $seq3` evaluates to the sequence containing A only

10 Arithmetic Expressions §3.4

`-expr +expr * div idiv mod + -`

`idiv` divides the first argument by the second, and returns the integer obtained by truncating the fractional part of the result.

`mod` returns the remainder resulting from dividing $arg1, the dividend, by $arg2, the divisor.

11 Comparison Expressions §3.5

Comparison expressions allow two values to be compared. The kinds of comparison expressions are **value**, **general**, and **node** comparisons.

`eq ne lt le gt ge = != < <= > >= is <<(precedes) >>(follows)`

Note: When an XPath expression is written within an XML document, the XML escaping rules for special characters must be followed; thus "<" must be written as "`<`".

Value Comparisons §3.5.1

`eq ne lt le gt ge`

Value comparisons are used for comparing single values. If the result of atomization is an empty sequence, the result of the comparison is an empty sequence. If the result of atomization is a sequence containing more than one value, a type error is raised.

- `$book1/author eq "Kennedy"` true only if the result of atomization is the value "Kennedy" as an instance of `xs:string` or `xs:untypedAtomic`.
- `//product[weight gt 100]` selects products whose weight is greater than 100. For any product that does not have a weight subelement, the value of the predicate is the empty sequence, and the product is not selected.
- `my:hatsize(5) eq my:shoesize(5)` true if my:hatsize and my:shoesize are both user-defined types that are derived by restriction from a numeric type.
- `fn:QName("http://example.com/ns1", "this:color") eq fn:QName("http://example.com/ns1", "that:color")`

General Comparisons §3.5.2

`= != < <= > >=`

General comparisons are quantified comparisons that may be applied to operand sequences of any length. The result of a general comparison that does not raise an error is always true or false.

- `$book1/author = "Kennedy"` true if the typed value of any author subelement of $book1 is "Kennedy" as an instance of xs:string or xs:untypedAtomic:
- `(1, 2) = (2, 3)` is true
- `(2, 3) = (3, 4)` is true
- `(1, 2) = (3, 4)` is false
- `(1, 2) = (2, 3)` is true
- `(1, 2) != (2, 3)` is true

Note: = and != operators are not inverses of each other.

- `$a, $b`, and `$c` are bound to element nodes of type annotation xs:untypedAtomic, with string values "1", "2", and "2.0" respectively. Then `($a, $b) = ($c, 3.0)` returns false because $b and $c are compared as strings, but, `($a, $b) = ($c, 2.0)` returns true, because $b and 2.0 are compared as numbers.

Node Comparisons §3.5.3

`is <<(precedes) >>(follows)`

Node comparisons are used to compare two nodes, by their identity or by their document order.

- The operands of a node comparison are evaluated in implementation-dependent order.
- If either operand is an empty sequence, the result of the comparison is an empty sequence.
- Each operand must be either a single node or an empty sequence; otherwise a type error is raised.
- A comparison with the `is` operator is true if the two operand nodes have the same identity, and are thus the same node; otherwise it is false. See [XQuery/XPath Data Model (XDM)] for a definition of node identity.
- A comparison with the << operator returns true if the left operand node precedes the right operand node in document order; otherwise it returns false.
- A comparison with the >> operator returns true if the left operand node follows the right operand node in document order; otherwise it returns false.
- `/books/book[isbn="1558604820"] is /books/book[call="QA76.9 C3845"]` true only if the left and right sides each evaluate to exactly the same single node
- `/transactions/purchase[parcel="28-451"] << /transactions/sale[parcel="33-870"]` true only if the node identified by the left side occurs before the node identified by the right side in document order.

12 Logical Expressions §3.6

`and or`

If a logical expression does not raise an error, its value is always one of the boolean values `true` or `false`.

- `1 eq 1 and 2 eq 2` is true
- `1 eq 1 or 2 eq 3` is true
- `1 eq 2 and 3 idiv 0 = 1` returns false or error in XPath 1.0 compatibility mode result is false
- `1 eq 1 or 3 idiv 0 = 1` returns true or error, in XPath 1.0 compatibility mode result is true
- `1 eq 1 and 3 idiv 0 = 1` returns an error

13 For Expressions §3.7

```
for $i in (10, 20),
    $j in (1, 2)
return ($i + $j)
```
result is a sequence of numbers: `11, 12, 21, 22`

A variable bound in a `for` expression comprises all subexpressions of the `for` expression that appear after the variable binding. The scope does not include the expression to which the variable is bound.

The following example illustrates how a variable binding may reference another variable bound earlier in the same `for` expression:

```
for $x in $z, $y in f($x)
    return g($x, $y)
```

The focus for evaluation of the `return` clause of a `for` expression is the same as the focus for evaluation of the `for` expression itself. Example:

- `fn:sum(for $i in order-item return @price * @qty)` find the total value of a set of order-items (incorrect)
- `fn:sum(for $i in order-item return $i/@price * $i/@qty)` find the total value of a set of order-items (correct)

14 Conditional Expressions §3.8

```
if ($widget1/unit-cost < $widget2/unit-cost)
  then $widget1
  else $widget2
if ($part/@discounted)
  then $part/wholesale
  else $part/retail
```

15 Quantified Expressions §3.9

`some every`

- `some`, the expression is true if at least one evaluation of the test expression has the effective boolean value true; otherwise the quantified expression is false.
- `every`, the expression is true if every evaluation of the test expression has the effective boolean value true; otherwise the quantified expression is false.
- `every $part in /parts/part satisfies $part/@discounted` `true` if every part element has a discounted attribute (regardless of the values of these attributes)
- `some $emp in /emps/employee satisfies ($emp/bonus > 0.25 * $emp/salary)` `true` if at least one employee element satisfies the given comparison expression
- `some $x in (1, 2, 3), $y in (2, 3, 4) satisfies $x + $y = 4` evaluates to true
- `every $x in (1, 2, 3), $y in (2, 3, 4) satisfies $x + $y = 4` evaluates to false
- `some $x in (1, 2, "cat") satisfies $x * 2 = 4` may either return true or raise a type error, since its test expression returns true for one variable binding and raises a type error for another
- `every $x in (1, 2, "cat") satisfies $x * 2 = 4` may either return false or raise a type error, since its test expression returns false for one variable binding and raises a type error for another

16 Expressions on SequenceTypes §3.10

Instance Of §3.10.1

The boolean operator `instance of` returns `true` if the value of its first operand matches the SequenceType in its second operand.

- `5 instance of xs:integer` returns `true`
- `5 instance of xs:decimal` returns `true` because `xs:integer` is derived by restriction from `xs:decimal`.
- `(5, 6) instance of xs:integer+` returns `true` because the given sequence contains two integers
- `. instance of element()` returns `true` if the context item is an element node or false if the context item is defined but is not an element node

Cast and Castable §3.10.2 and §3.10.3

The expression `V castable as T` returns `true` if the value `V` can be successfully cast into the target type `T` by using a `cast` expression; otherwise it returns `false`. The `castable` expression can be used as a predicate to avoid errors at evaluation time. It can also be used to select an appropriate type for processing of a given value, as illustrated in the following example:

```
if ($x castable as hatsize)
   then $x cast as hatsize
   else if ($x castable as IQ)
   then $x cast as IQ
   else $x cast as xs:string
```

Note: If the target type of a `castable` expression is `xs:QName`, or is a type that is derived from `xs:QName` or `xs:NOTATION`, and the input argument of the expression is of type `xs:string` but it is not a literal string, the result of the `castable` expression is `false`.

Constructor Functions §3.10.4

The name of the constructor function is the same as the name of its target type (except `xs:NOTATION` and `xs:anyAtomicType`) including namespace. The constructor function call `T($arg)` is defined to be equivalent to the expression `(($arg) cast as T?)`.

The constructor functions for `xs:QName` and for types derived from `xs:QName` and `xs:NOTATION` require their arguments to be string literals or to have a base type that is the same as the base type of the target type; otherwise a type error is raised.

- `xs:date("2000-01-01")` is equivalent to `("2000-01-01" cast as xs:date?)`
- `xs:decimal($floatvalue * 0.2E-5)` is equivalent to `(($floatvalue * 0.2E-5) cast as xs:decimal?)`
- `xs:dayTimeDuration("P21D")` returns a xs:dayTimeDuration value equal to 21 days. It is equivalent to `("P21D" cast as xs:dayTimeDuration?)`
- `usa:zipcode("12345")` is equivalent to the expression `("12345" cast as usa:zip-code?)`

An instance of an atomic type that is not in a namespace can be constructed in either of the following ways:

- `17 cast as apple`
- `apple(17)`

Treat §3.10.5

`treat` can be used to modify the static type of its operand.

Like `cast`, the `treat` expression takes two operands: an expression and a SequenceType. Unlike `cast`, however, `treat` does not change the dynamic type or value of its operand. Instead, the purpose of `treat` is to ensure that an expression has an expected dynamic type at evaluation time.

- `$myaddress treat as element(*, USAddress)` at run-time, the value of `$myaddress` must match the type `element(*, USAddress)`