XML and Databases

Chapter 11: XPath II: Expressions

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Winter 2022/23

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

Objectives

After completing this chapter, you should be able to:

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

Contents

Lexical Syntax (1)

- XPath has no reserved words. Thus, there are no restrictions for element names.
- The context helps to detect special names:
  - Axes are followed by "`::`".
  - Functions, sequence types, `if`: followed by "`(`".
  - `for`, `some`, and `every` are followed by "`$`".
  - Operators such as "`and`" are distinguished from element names by the preceding symbol (Is a continuation with an element name possible?).

    Some "keywords", e.g. "`cast as`", deliberately consist of two parts.

Lexical Syntax (2)

- Some more ambiguities:
  - If a name immediately follows `/`, and is not followed by `::`, it is assumed that it is an element name.

    Thus, in `/ union /*`, the word "`union`" is an element type name. If one wants the ∪-operator, one must write `(/) union /*`.

  - If `+`, `*`, `?` follow a sequence type, it is assumed that they are an occurrence indicator (belonging to the type).

    E.g. `4 treat as item() + - 5` is implicitly parenthesized as `(4 treat as item()+) - 5`, not as `(4 treat as item()) + -5`.

Lexical Syntax (3)

- Variable names are marked by prefixing them with “`$`”, e.g. “`$x`”, “`$p:x`” (a variable name is a `QName`).

  XPath 2.0 allows whitespace between “`$`” and the QName, 1.0 not.

- Note that in contrast to some interpreted languages, variables are not simply replaced by their value, before the expression is parsed.

  - E.g. even if `$x` has the value “`BOOK`”, `//$x` does not mean `//BOOK`, but gives a type error.

    One has to use `//*[local-name(.)=$x]`.

Lexical Syntax (4)

- Whitespace is possible between each two tokens.
- The next token is always the longest sequence of characters that can comprise a token.

  This is the usual rule in programming languages.
- E.g. `x-1` is only a single XML name (names can contain hyphens). If one wants "the value of child element `x` minus 1" one must use spaces: `x - 1`.

  The space before the "`1`" is not necessary: an integer literal contains no sign (but there is a unary "`-`"). Note that "`x+1`" is possible without spaces (XML names cannot contain "`+`").

Lexical Syntax (5)

- There are three types of numeric literals:
  - A sequence of digits , e.g. “`123456`”, has type `xs:integer`.
  - A sequence of digits containing a single “`.`”, e.g. “`12.34`”, has type `xs:decimal`.

    The “`.`” can be at the beginning, e.g. “`.3`”, at the end, e.g. “`1.`”, or somewhere between the digits, e.g. “`3.14159`”.
  - A number in scientific notation, e.g. “`1.2E-7`”, or “`1e9`” or “`.3E+8`”, has type `xs:double`.
- In XPath 1.0, all numeric literals had type `double`.

Lexical Syntax (6)

- A string literal is
  - a sequence of characters enclosed in `'`, or
  - a sequence of characters enclosed in `"`.
- If the delimiter appears within the sequence, it must be doubled, e.g. `'Stefan''s'`.

  The possibility to include the string delimiter by doubling it is new in XPath 2.0.
- Special characters (other than the delimiters) can be included in the string by using the escaping mechanism of the host language, e.g. character or entity references in XML.

Lexical Syntax (7)

- XPath is used in XSLT as XML attribute values.
- Then character and entity references are expanded before the XPath processor sees the input.

  Thus, it does not help to use an entity reference to include the string delimiter in the string literal. This was probably the reason for using a different mechanism than XML uses for attribute values: There the doubling is not supported, one must use an entity/character reference. Of course, if the delimiter of the XML attribute value that contains the XPath expression is used inside the XPath expression, it must be written as a character or entity reference. E.g. `select="'&quot;'''"` contains the XPath expression `'"'''`, which yields the string `"'`.
- Also, whitespace in attribute values is normalized.

  XPath sees only a single space. Use character or entity references.

Lexical Syntax (8)

- Constructor functions can be used to denote constant values of other types, e.g.

  `xs:date("2007-06-30")`

  The string must use the lexical syntax defined in XML Schema.

- This can also be used for special floating point values, e.g. positive infinity (result of an overflow):

  `xs:double("INF")`

- The boolean values can be written as calls to the built-in functions `true()` and `false()`.

Lexical Syntax (9)

- Comments are delimited in XPath with smilies "`(:`" and "`:)`", e.g.

```
(: This is a comment :)
```

  Comment delimiters known from other languages did not work in XPath. E.g. `/*` and `//` have already an important meaning in XPath, `--` can appear in XML names. The end of line is removed by attribute value normalization. Braces `{...}` are used in XSLT for attribute value templates, and have an important role in XQuery.

- Comments can be nested.

  Thus, one can "comment out" a section of code that itself contains a comment. Note however, that when the lexical scanner is in "comment mode", it ignores the beginning of string constants. Thus `(: ":)" :)` gives a syntax error, although `":)"` in itself is ok.

Contents

Accessing the Context

- The context item is written as "`.`".

  This is also new in XPath 2.0. In XPath 1.0, "`.`" was only an abbreviation for "`self::node()`".

- The context position is returned by the built-in function `position()`.

  When iterating over a sequence, the first item has the position 1 (not 0 as in C-style arrays).

- The context size is returned by the built-in function `last()`.

Sequence Constructor (1)

- The comma operator "`,`" is used as sequence constructor, e.g. `1, 2` is the sequence consisting of `1` and `2`.
- Formally, `E1, E2` is the concatenation of sequences `E1` and `E2`.

  Remember that in XDM everything is a sequence, even the numbers `1` and `2` in the previous example are formally identified with the corresponding singleton sequences. Vice versa, one could also say that `E1, E2` first constructs a sequence of length 2 with (the values of) `E1` and `E2` as items, but since sequences can never contain other sequences, the result is then flattened.

Sequence Constructor (2)

- Since the comma is also used for other syntactic purposes (e.g. in the function argument list), the expression `E1, E2` must be enclosed in parentheses `(...)` in many contexts.

  The formal grammar has a symbol "`exprSingle`" that is an arbitrary expression, but without "`,`" on the outermost level.

- `()` denotes the empty sequence.

- Note the flattening rules. E.g. `(1, (), (2, 3))` is a legal expression, but it evaluates to `(1, 2, 3)`.

  In XDM, sequences can never contain other sequences.

Numeric Range Constructor

- m `to` n generates the sequence of integers from m to n (inclusive).

  If $n \le m$, the result is the empty sequence. The arguments m and n must be integers, or belong to a subtype of integer, or be untyped and convertable to integer. If one of the arguments is of another type (or is the empty sequence), an error occurs.

- E.g. `1 to 5` generates `(1, 2, 3, 4, 5)`.
- A good implementation will not actually materialize the complete sequence, but instead construct a loop over the elements (“lazy construction”).

Set Operations

- `E1 | E2` returns the union of the sequences E1 and E2. One can equivalently write `E1 union E2`.

  The input sequences must consist of nodes only, or a type error is raised. The result is a sequence of nodes in document order without duplicates (the closest a sequence can come to a true set). These rules also apply to the other set operations `intersect` and `except`.

- `E1 intersect E2` returns the set of nodes that are contained in both, E1 and E2.

- `E1 except E2` is the set of nodes that occur in E1, but not in E2.

  `intersect` and `except` have equal priority. They bind stronger (have higher priority) than `union` and `|`.

Contents

Atomization (1)

- In contexts where atomic values are needed (e.g., in the arguments to arithmetic operators), XPath applies a type coercion called “atomization”.
- It also has a built-in function `data(`*s*`)` that returns the result of applying atomization to the input sequence *s*.
- For example, consider (`ge` means $\geq$):

```
//RESULT[@POINTS ge 8]
```

- `@POINTS` selects an attribute node, but for the comparison, its value (an integer) must be determined.

Atomization (2)

- The result of atomization is computed by looping over the input sequence:
  - If the current list item is an atomic value, it is appended to the output sequence.
  - If the current list item is a node that has a typed value, this typed value is appended to the output.

    The typed value might consist of zero, one, or more atomic values.
  - Otherwise (node with typed value undefined), an error is raised.

    This happens only for elements that are declared with pure element content, when they were validated against a schema.

Comparison Operators (1)

- XPath has three kinds of comparison operators:
  - Value comparison operators: `eq`, `ne`, `lt`, `le`, `gt`, `ge`.
  - Node comparison operators: `is`, `<<`, `>>`.
  - General comparison operators: `=`, `!=`, `<`, `<=`, `>`, `>=`.
- XPath 1.0 had only the general comparison operators.

  The behaviour of these operators can sometimes cause surprises, and makes optimization difficult. Therefore, a safer set of operators was introduced in XPath 2.0.
- Note that when XPath expressions appear in XML attribute values, "`<`" must be written "`&lt;`".

Comparison Operators (2)

Value Comparison Operators (`eq`, `ne`, ...):

- Type checking for value comparison:
    - First, atomization is applied to both operands. Let the result be x and y.
    - If x or y is a sequence consisting of more than one item, a type error occurs.
    - If x or y is the empty sequence, the result is the empty sequence (later treated like false).
    - `untypedAtomic` is converted to `string`.
    - Derived types are converted to the base type.
    - Now the types must be identical, or both must be numeric. Otherwise a type error occurs.

Comparison Operators (3)

Value Comparison Operators (`eq`, `ne`, . . .), continued:

- For documents that are not validated against a schema, one must use explicit type conversions.
- E.g.. if the typed value of `@POINTS` has the type `untypedAtomic`, a comparison like

  `@POINTS ge 8`

  generates a type error, because `8` is an `integer`, and `@POINTS` is converted to a `string`.

  Note that e.g. `@FIRST eq "Ann"` would work.

  Solution: use “`number(@POINTS)`” or “`xs:integer(@POINTS)`”.

  Of course, validating the document against a schema would be better.

Comparison Operators (4)

Value Comparison Operators (eq, ne, ...), continued:

- Meaning of value comparison operators:
  - eq: equal ($=$).
  - ne: not equal ($\neq$).
  - lt: less than ($<$).
  - le: less than or equal ($\leq$).
  - gt: greater than ($>$).
  - ge: greater than or equal ($\geq$).
- For details, please look into the standard.

  E.g. for date and time types, the implicit timezone is used, instead of the partial order that XML Schema defines.

Comparison Operators (5)

Node Comparison Operators (`is`, `<<`, `>>`):

- Both operands must be a single node or the empty sequence (else a type error occurs).
- If one is the empty sequence, the result is the empty sequence (often treated like false).
- *x* `is` *y* is true if *x* and *y* are the same node.
- *x* `<<` *y* is true if *x* comes before *y* in document order.

  If the nodes are in different documents, the order is implementation dependent, but stable.
- *x* `>>` *y* is true if *x* comes after *y*.

Comparison Operators (6)

General Comparison Operators (=, !=, ...):

- Both operands are atomized, yielding sequences x and y of atomic values.
- Now all possible combinations of $x_i \in x$ and $y_j \in y$ are compared according to the rules on the next slide.
  If one comparison yields true, the result is true.
  If all the single comparisons return false, the result is false.

  Actually, a comparison might also generate a runtime error (type error). If the runtime error happens before a comparison yields true, the result is the runtime error. If the processor detects the true value first, it will most probably not do any further comparisons. One cannot rely on any particular order of the comparisons.
- I.e. there is an implicit existential quantifier over x and y.

Comparison Operators (7)

General Comparison Operators (=, !=, . . .), continued:

- If x_i and y_j are both of type `untypedAtomic`, they are converted to `string`.
- If one, e.g. x_i, is of type `untypedAtomic` and the other (y_j) is of a more specific type, x_i is converted to the type of y_j.

  Unless the type of y_j is numeric, then `double` is chosen for x_i. E.g. if x_i is `0.3`, and y_j is the integer 0, this rule ensures that x_i is not converted to an integer.
- After these conversions, x_i and y_j are compared with the corresponding value comparison operator (e.g. `eq` for `=`).
- Note that with a general comparison operator, `@POINTS >= 8` is ok even if the document is not validated.

  The right operand is of type `integer`, therefore the left operand is converted from `untypedAtomic` to `double` and a numeric comparison is done.

Comparison: Surprises (1)

- In XPath 1.0,
  - `1 = true()`,

    When comparing a number with a boolean value, the number is first converted to a boolean: Every number except 0 and NaN becomes true. (The priority list of types for `=`/`!=` comparison in XPath 1.0 is boolean, number, string.)
  - `true() = "true"`,

    When comparing a string with a boolean value, the string is converted to boolean. Every string except `""` is converted to true.
  - `1 != "true"`, i.e. the transitivity of `=` is violated!

    When comparing a string and a number, the string is converted to a number. In this case, `"true"` is converted to `NaN`.
- In XPath 2.0, these are all type errors.

Comparison: Surprises (2)

- However, such a situation can also be constructed in XPath 2.0 when no schema validation was done:
  - Let the context node be

    `<X A="1" B="1.0"/>`
  - `@A = 1` is true,

    `@A` has type `untypedAtomic`, thus a numeric comparison is done: `@A` is converted to `double`, then `1` is also converted to `double`.
  - `1 = @B` is true,

    As above, a numeric comparison is done.
  - `@A = @B` is false (transitivity is violated).

    If both operands have type `untypedAtomic`, then a string comparison is done (both are converted to `string`).

Comparison: Surprises (3)

- The implicit existential quantification in the general comparison operators can cause surprises:
  - `$x != 1` and `$x = 1` can be true at the same time.

    E.g., consider `$x = (1, 2)`. This also shows that `$x != 1` is not the same as `not($x = 1)`. In this example, `not($x = 1)` is false.
  - `$x = $x` does not always hold.

    If `$x` is the empty sequence, the implicit existential quantification is obviously false, even if the quantified condition is a tautology.
  - Transitivity of `=` and other relations can be violated even in schema validated documents.

    E.g. `(1) = (1,2)` and `(1,2) = (2)` are true, but `(1) = (2)` is false.

Exercise (1)

```
<?xml version="1.0"?>
<BOOKLIST>
    <BOOK ISBN="0-13-014714-1" PAGES="1074">
        <AUTHOR FIRST="Paul" LAST="Prescod"/>
        <AUTHOR FIRST="Charles" LAST="Goldfarb"/>
        <TITLE>The XML Handbook - 2nd Edition</TITLE>
        <PUBL DATE="19991112">Prentice Hall</PUBL>
        <NOTE>Contains CD.</NOTE>
    </BOOK>
    <BOOK ISBN="1-56592-709-5" PAGES="107">
        <AUTHOR FIRST="Robert" LAST="Eckstein"/>
        <TITLE>XML Pocket Reference</TITLE>
        <PUBL DATE="19991001">O'Reilly</PUBL>
    </BOOK>
</BOOKLIST>
```

Exercise (2)

- What will be the result of this expression?

  `/BOOKLIST/BOOK[AUTHOR/LAST="Goldfarb"]`

- Would this work with "`eq`" instead of "`=`"?

  "`/`" binds stronger (has higher priority) than "`=`" and "`eq`".

- Please write an XPath expression for:
  - Print the last names of the author of the "XML Pocket Reference" (book title).

    Assume that the context node is the document node and that it suffices to select the attribute nodes, and not necessarily take their value.

Contents

Arithmetic Operators (1)

- +: Addition

  The arithmetic operators and numeric functions (see below) have four versions with signature $T \times T \rightarrow T$, where T is one of: `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`. Of course, one can also substitute a derived type for one of these types, but the result will be the base type. E.g., if one adds two values of type `xs:positiveInteger`, the result is of type `xs:integer`. Furthermore, type promotion is done: If values of two different numeric types are added, the one earlier in the above list is converted to the one later in the list, e.g. for `1 + 2e3`, the value 1 (of type `xs:integer`) is converted to `xs:float`, and then a floating point addition is done. In XPath 1.0, all numbers were considered as `double` values.

- -: Subtraction

  The operators `+` and `-` exist in unary and in binary form. The unary `+` is new in XPath 1.0 (it was added for compatibility with XML Schema).

Arithmetic Operators (2)

- `*`: Multiplication
- `div`: Division

  The symbol `/` could not be used (otherwise: ambiguous path expressions). As an exception to the signature $T \times T \rightarrow T$, the result type for integer operands is `xs:decimal`. The other three cases are as usual.
- `idiv`: Integer Division

  This operator exists with signatures $T \times T \rightarrow$ `xs:integer` where T is one of the four numeric types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`. The result of division is truncated, e.g. `9 idiv 5 = 1`.
- `mod`: Remainder of the integer division (modulo)

  This has again signature $T \times T \rightarrow T$. Except for error conditions and special floating point values, `(`x `idiv` y`) *` y `+ (`x `mod` y`) =` x holds.

Contents

Logical Conditions (1)

- and: Conjunction (both operands must be true).

  The effective boolean value of the operands is automatically determined. For instance, `()`, `""`, `0` are treated like false. A sequence that starts with a node, a non-empty string, and a non-zero number (except `NaN`) are treated like true.

  Note that atomization is not applied to the operands. So an attribute node is treated like true, even if its value is the boolean value false. One could explicitly call `data(...)` or do a comparison.

  In XPath 1.0, it was guaranteed that the right operand was evaluated only if the left operand was true. In XPath 2.0, this is no longer guaranteed, so that the query optimizer gets more freedom (e.g., there might be an index for the condition on the right side). However, one can use an `if`-expression to avoid possible run-time errors. Basically, `A and B` is equivalent to `if A then B else false()`.

- or: Disjunction (at least one operand is true).

Logical Conditions (2)

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

  Formally, this is a function without parameters, that always returns the value "true". Because XPath has no reserved words, the parentheses are necessary to remove the ambiguity (see Slide 59).

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

- `not(`*C*`)`: Negation of condition *C*.

  Again, this is formally a function, not an operator (so the parentheses are necessary). The function mainly translates true to false and false to true. However, before this, it automatically computes the effective boolean value of the argument. So the argument of the function is declared as an arbitrary sequence (`item()*`), not as `xs:boolean`. However, certain inputs can generate a type error (see "effective boolean value" in Chapter 10).

- `=`, `<`, . . . can be used on boolean values (false<true).

Logical Conditions (3)

- An existential quantifier (∃, “there is”) over a sequence is written as

  `some` v `in` S `satisfies` C

  where
  - v is a variable (starting with “`$`”)
  - S is an expression that generates a sequence of values that are assigned to v one by one,
  - C is an expression, of which the effective boolean value is determined for each such variable assignment: If it is true for at least one assignment, the value of the entire `some`-expression is true.

Logical Conditions (4)

- For instance, the following is true:

  `some $i in (1, 2, 3) satisfies $i > 2`

- A universal quantifier (∀, "for all") over a sequence is written as

  `every` v `in` S `satisfies` C

- If the binding sequence S should be empty,
  - `some` is false (there is no satisfying assignment)
  - `every` is true (no counterexample can be found)

- Note that the focus is not changed when C is evaluated. Thus, it (more or less) must contain v.

Logical Conditions (5)

- Nondeterministic outcome for runtime errors:
  - An implementation can check the different variable assignments in an arbitrary order.
  - It can also stop as soon as the truth value of the entire expression is clear.

    I.e. when it found one value in S for which the `some`-quantified condition C was true, it is clear that the `some`-expression is true. In the same way, if C was false once, an `every`-condition is false.
  - If the evaluation of C for some assignment would cause a runtime error, but the evaluation stops before this assignment, one cannot rely on the fact that this will always be the case.

Logical Conditions (6)

- One can quantify several variables in a single `some` or `every` expression:

  `some` v_1 `in` S_1`, ..., `v_n `in` S_n `satisfies` C

- Then conceptually all possible combinations of values are tested (e.g., in a nested loop).

  As explained above, it can stop earlier, if the result is clear.

- If S_i or C use the comma-operator, it must be inside parentheses.

- The scope of v_i includes S_j for $j > i$ and C (i.e. the entire rest of the expression after S_i can use v_i).

Exercise (1)

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENT>
    <SID>101</SID>
    <FIRST>Ann</FIRST>
    <LAST>Smith</LAST>
  </STUDENT>
  ...
  <RESULT>
    <SID>101</SID>
    <CAT>H</CAT>
    <ENO>1</ENO>
    <POINTS>10</POINTS>
  </RESULT>
  ...
```

Exercise (2)

- Please write the following queries in XPath:
  - What is the SID of Ann Smith?

    It suffices that the SID element is selected. If necessary, one can explicitly call `data(...)` to perform an atomization.
  - Please print the last names of all students who got more than 8 points for Homework 1.

    Note that this exercise already requires a (semi-)join. One can apply the `some`-quantifier to get a name for one of the needed nodes, and use the context/focus for the other node.
- What is the error in

```
//EXERCISE[some $r in //RESULT satisfies
                               POINTS = MAXPT]
```

Contents

For Expressions (1)

- The `for`-Expression can be used to map every element of an input sequence to zero, one or more elements of an output sequence:

  for v in S return E

- The variable v is bound to each element of the input sequence S in turn, and the expression E is evaluated. The resulting sequences are concatenated. For example:

```
for $i in (1, 2, 3) return $i * 10
```

  returns (10, 20, 30).

For Expressions (2)

- I.e. in the expression

  `for v in S return E`

  the variable *v* loops over the sequence *S*, and in each iteration, the result of evaluating the expression *E* is appended to the output sequence.

  Often, the expression *E* will evaluate to single values (sequences of length 1), then each element in the input sequence is mapped to the element in the output sequence at the same position.

  Of course, *E* nearly always contains variable *v*. Note that the context position is not changed during the iteration. Only *v* changes.

- `for` can be nicely combined with the numeric range constructor, e.g.: `for $i in 1 to 3 return $i*10`

For Expressions (3)

- One can also let several variables run over different sequences, then all combinations are considered:

  `for` v_1 `in` S_1`,` v_2 `in` S_2 `return` E

- A typical implementation are nested loops, but the query optimizer can of course choose a different, more efficent evaluation strategy.

  But the order in the output sequence cannot be changed, unless this is input for a function that does not need a specific order (e.g., `count`).

- The above expression is equivalent to

  `for` v_1 `in` S_1 `return (for` v_2 `in` S_2 `return` E`)`

For Expressions (4)

- `for` binds stronger than the comma operator (sequence constructor). Thus, if S or E contain the comma operator, it must be inside parentheses.
- In `for` v_1 `in` S_1`,` v_2 `in` S_2 `return` E

  the scope of the variable v_1 consists of S_2 and E. The scope of variable v_2 consists only of E.

  I.e. one can use v_1 when defining the values for v_2. This is compatible with the nested version of a `for`-expression with several variables.
- `for`-expressions are a simplified version of FLWR-expressions in XQuery. They are new in XPath 2.0.

For Expressions (5)

- The path expression `book/author` is equivalent to

  `for $b in book return $b/author`

- In general, differences between `/` and `for` are:
  - `/` uses the implict context, `for` explicit variables.

    `for` can use several variables, `/` has always only one context item.
  - `/` works only on nodes, `for` on arbitrary data.
  - `/` sorts the result in document order and eliminates duplicates, `for` does not do this.

If Expressions (1)

- The expression

  $$\texttt{if(}C\texttt{) then } E_1 \texttt{ else } E_2$$

  is evaluated as follows:

  - First, the effective boolean value of C is determined (no atomization is done).
  - If the effective boolean value of C is true, E_1 is evaluated, and its value is the value of the entire `if`-expression.

    It is guaranteed that E_2 is not evaluated in this case.

  - Otherwise, the value of E_2 is returned.

    In this case, E_1 is not evaluated.

If Expressions (2)

- The guarantee that the other branch is not evaluated is important if it could cause a runtime error.
- If the expressions E_1 or E_2 contain the comma operator, it must be inside parentheses.

  Since there is no “`fi`” (or “`end if`”), a comma in E_2 could cause an ambiguity, when the expression is used in a function call. In E_1 it would be no problem, but there it is excluded for reasons of symmetry.
- Note that the `else`-part is not optional. One often sees “`else ()`”.

  This avoids the “dangling else” ambiguity that occurs in many programming languages.

Operator Precedences (1)

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

(continued on next slide)

Operator Precedences (2)

(continued from previous slide)

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

Summary: New Constructs

- The following constructs are new in XPath 2.0:
  - `for`, `some`, `every`, `if`, `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `is`, `<<`, `>>`, `intersect`, `except`, `idiv`, `to`, `,`

    In XPath 1.0, no variables could be bound inside the expression (only variables declared in the XSLT context could be used).
  - Function calls in path expressions.
  - A much richer type system (conformant with XML Schema), stricter type checking.

    XPath 1.0 had only four data types: node set, boolean, number, string. XPath 2.0 can also work with user-defined types.
  - Arbitrary sequences instead of node sets.
  - A much larger function library (see next section).

Syntax: Surprise (1)

- The following XPath expression is legal:

  `for div div`

- E.g., if the context node is

  `<X><for>8</for> <div>2</div></X>`

  the result is 4 or 4.0 ($= 8/2$).

  The expression consists of the operator `div`, applied to the results of the path expressions `for` (left operand) and `div` (right operand). The path expression `for` returns the child node with name `for`. Since this is input to `div`, it is atomized, this results in the value 8 or 8.0 (if `for` is declared with simple content of a numeric type, or if the document was no schema-validated: Then the value is `"8"`, but of type `untypedAtomic`, so it can be converted to the number `8.0`).

Syntax: Surprise (2)

- The following XPath expression is legal:

```
***
```

- Exercise: What is the result if the context node is

```
<X><Y>3</Y></X>
```

Exercise

What is the meaning of:

- `@WEEKDAY = ('Sat', 'Sun')`
- `$x idiv 1`
- `(@QUANTITY, 1)[1]`
- `if @GUEST = true() then... else ...` vs.
  `if @GUEST then... else ...`
- `true` vs. `true()`
- `not(*)`
- `not /A` vs. `not(/A)`

Contents

Type Casts (1)

- For some pairs of types T_1 and T_2, some values v_1 of type T_1 can be converted to a value v_2 of type T_2.
- For instance, if T_1 is `xs:string` or `xs:untypedAtomic`, and v_1 conforms to the lexical representation of T_2 as defined in the XML Schema Standard, then the conversion is possible.

  Special restrictions apply for target types `xs:NOTATION` (XML Schema states that only subtypes of it can be instantiated) and `xs:QName` (only string literals can be converted, and only if they use a namespace prefix from the static context or the default namespace).

Type Casts (2)

- The conversion is written

  v_1 `cast as` T_2 e.g. `"123" cast as xs:integer`

- Using a constructor function is equivalent, except that the constructor function can map `()` to `()`:

  T_2`(`v_1`)` e.g. `xs:integer("123")`

  This works also for user defined types. But the default namespace of the two variants differs. For functions, including constructor functions, the default namespace is `http://www.w3.org/2005/xpath-functions`. For the `cast as` syntax, the default namespace is the same as used for element types. The argument type of the constructor function is `anyAtomicType?`, the result type is T_2`?`.

- Exact equivalent: v_1 `cast as` T_2`?`

Type Casts (3)

- There is a special constructor function that constructs a `xs:dateTime` value from an `xs:date` and an `xs:time` value.
- One can cast only to atomic types, possibly with the occurrence indicator "`?`".

  This means that one cannot cast to list or union types, as well as more general sequences.
- One cannot cast to `anyAtomicType`, because at runtime, there are no values of this type.

  Of course, `untypedAtomic` is possible.

Type Casts (4)

- Atomization is applied to the argument of the cast-expression or the constructor function.

  Thus, one can e.g. use a path expression that selects an attribute node. The value of that node is taken automatically.

- If the result is a sequence of two or more values, an error is raised.
- An error also occurs if the value cannot be converted, e.g. the string does not have the right format.

  This error may occur as a static error if the argument is e.g. given as a string literal, or as a runtime error, when the value is not known at compile time.

Type Casts (5)

- All sensible type conversions are supported, not only conversions from string.

  E.g. arbitrary conversions between numeric types are possible, as long as the value fits into the result type (for floating point types, even that is no problem, since they have the special values `INF` and `-INF`). The complete list is given in the specification “XQuery 1.0 and XPath 2.0 Functions and Operators”, Section 17.1.

- When casting to a derived type, the value is first converted to the corresponding base type, and then the constraining facets are checked.

  E.g. if `money` is a type derived from `xs:decimal` with `fractionDigits=2`, one cannot convert the value `1.234` to `money`. However, as an exception, values can be converted to `xs:integer` by truncation.

Type Casts (6)

- Since *v* `cast as` *T* can cause a runtime error, XPath also offers the condition

  v `castable as` *T*

- This condition is true if and only if the cast would succeed without error.
- Thus, one can use an `if`-expression to handle the case that the value cannot be converted.

Exercise

- Name (at least) two cases, where the following function calls differ:
  - `boolean(`*v*`)`: This computes the effective boolean value of *v*.
  - `xs:boolean(`*v*`)`: Constructor function, does first atomization.
- What happens if an integer needs to be converted to a subtype of `xs:decimal` with a pattern that prescribes two digits after the decimal point?

  When converting to a derived type with the `pattern`-facet, only the canonical representation of the value is checked.

Runtime Type Check (1)

- XML Schema supports union types, e.g. `grade_t` might be the union of the string values `"passed"`, `"failed"`, and integer values from `1` to `5`.

  1: “very good”, 2: “good’, 3: “satisfactory”, 4: “fair”, 5: “poor”.

- XDM permits only sequences of atomic values and nodes (it has no explicit support for union types): At runtime, the exact type of each value is known.

- If `GRADE` is an attribute of type `grade_t`, the value of this attribute will be a string or a number.

  Which of the two, is only known at runtime for a concrete instance.

Runtime Type Check (2)

- When the type of a value is not known at compile time, it must be tagged with a type identification at runtime.

  This is nothing else than the standard implementation of a union type. Thus, the fact that XDM has no explicit support for union types, does not mean much. Unknown types can also occur when a subtype is substituted for the supertype. A value might actually be of a subtype, but at compile time, only the supertype is known. Again, type tagging is used (e.g., the "virtual function table" in C++).

- Simple XPath implementations will tag every value in this way, and do all the type checking at runtime.

Runtime Type Check (3)

- In order to check whether an exam was passed, one might try the following condition (wrong!):

```
@GRADE = "passed" or @GRADE <= 4
```

- However, one cannot compare strings and integers:
  - If `@GRADE` is a string, the right condition gives a type error.

    If the left condition is true, and if that is checked first, the error might not occur, because the right condition is not evaluated.

  - If it is an integer, the left part gives a type error.

    If the right condition is checked first, it is possible that the error does not occur.

Runtime Type Check (4)

- Thus, XPath has the possibility to check the type of a value at runtime:

  v `instance of` T

  is true if value v has type T.

- In contrast to `cast as` and `castable as`, the type T may be any sequence type.

- Note that the condition is also true if the type tag of v is a type derived from T.

- For instance, the following is true:

  ```
  5 instance of xs:decimal
  ```

Runtime Type Check (5)

- However, `instance of` does not check whether a value happens to satisfy the constraints of a subtype. It only checks the type tag.
- For instance, the following is false:

  ```
  5 instance of xs:positiveInteger
  ```

  Numeric literals that consist entirely of digits are assigned the type `xs:integer`.
- Of course, the following is true:

  ```
  5 castable as xs:positiveInteger
  ```

Runtime Type Check (6)

- With instance of, one can write the condition as

```
if(@GRADE instance of xs:string)
then @GRADE = "passed"
else @GRADE <= 4
```

- This will work in a system based entirely on runtime type checking.
- In a system using static type checking ("at compile time"), it will probably still give a type error because the (not very intelligent) system does not understand that the comparisons are safe.

Static Type Checking (1)

- Some XPath implementations do all type checking at runtime, some try to do as much as possible at compile time ("static type checking").
- Advantages of static type checking:
  - Type errors that occur only sometimes cannot be found reliably with testing. Static type checking finds them.
  - Runtime is reduced (most tests done at compile time), memory too (fewer type tags).
  - Query optimzation can be improved.

Static Type Checking (2)

- “static type checking is a mixed blessing. It will report some errors early, but it will also report many false alarms. The more you are dealing with unpredictable or semi-structured data, the more frequent the false alarms will become. With highly structured data, static type checking can be a great help in enabling you to write error-free code; but with loosely structured data, it can become a pain in the neck.” [Michael Kay, 2004]
- Static type checking is the pessimistic assumption that what can go wrong, will go wrong.

Type Assertions (1)

- The expression

  `v treat as T`

  checks whether *v* has type *T* (or a subtype of *T*), then it returns *v* (unchanged). Otherwise, it causes a runtime error.

  In “`treat as`” the type can again be an arbitrary sequence type. E.g., one can also check whether a node is an element node.

- This expression is used when the compiler cannot derive that expression *v* has the dynamic type *T*, but the programmer wishes to assert that this will always be the case.

Type Assertions (2)

- Of course, the static type of "*v* `treat as` *T*" is *T*.

  The dynamic type of the value of an expression is always a subtype (or identical to) the static type of that expression (type safety).

- In the above example, the check whether an exam was passed can be written as follows to satisfy any static type checker:

```
if(@GRADE instance of xs:string)
then (@GRADE treat as xs:string) = "passed"
else (@GRADE treat as xs:integer) <= 4
```

- One can also use functions to make assertions on the length of sequences, see next slide.

Type Assertions (3)

- `exactly-one(s)`: Sequence s has length 1.

  Argument: `item()*`. Result: `item()`. If s consists of exactly one element, s is returned unchanged (one could also say that this element is returned, because XPath makes no difference between a sequence of length 1 and its element). If s is empty or consists of more than one element, a runtime error occurs. New in XPath 2.0.

- `one-or-more(s)`: Sequence s has length ≥ 1.

  Argument: `item()*`. Result: `item()+`. If s is empty, a runtime error occurs. Otherwise, it is returned unchanged. New in XPath 2.0.

- `zero-or-one(s)`: Sequence s has length ≤ 1.

  Argument: `item()*`. Result: `item()?`. If s is empty or consists of at exactly one element, s is returned unchanged. If s consists of more than one element, a runtime error occurs. New in XPath 2.0.

References

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