Chapter 6: XPath

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 January 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.
- Henning Behme: Mutabor (XSLT-Tutorial I: Grundlagen und erste Beispielanwendung). iX 1/2001, S. 67. [http://www.heise.de/ix/artikel/2001/01/167/]
- Miloslav Nic, Jiri Jirat: XPath Tutorial. Zvon [http://www.zvon.org/xxl/XPathTutorial/General/examples.html]
- Tracey Roy (Author), Dan Mabbutt (Editor): XSLT & XPath Tutorial. TopXML. [http://www.topxml.com/xsl/tutorials/intro/default.asp]

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.

Overview

1. Introduction, Software
2. Location Paths
3. Expressions
4. Data Types
5. XPath Functions

Introduction (1)

- XPath ("XML Path Language") is a standard for expressions that are mainly used for selecting parts of XML documents (nodes in the XDM tree).

  One can view this important subset of XPath as a pattern language: A tree node matches a pattern if it is contained in the result of evaluating the XPath expression (a sequence of nodes).

- However, XPath expressions can also compute atomic values or more generally any sequence allowed by XDM.

  Arithmetic expressions map numbers to numbers, XPath maps a set of documents (or really a "context", see below) to a sequence of nodes and atomic values. So it does not seem to be closed.

Introduction (2)

- XPath is used e.g. in
  - ◇ XSLT (XML Stylesheet Lang./Transformations)

    E.g. for defining to which nodes a transformation template should be applied, which parts of the input document should be copied to the output document, and where processing in the input document should continue after a template was applied.
  - ◇ XPointer

    To permit references to a part of a document. With classic URIs (plus "#..."), one can point only to places in an HTML document, where the author of the document has placed an anchor.
  - ◇ XML Schema

    For selecting nodes that are uniquely identified etc.
  - ◇ XQuery (XML Query Language)

Introduction (3)

- The reason for the name "XPath" is that the expressions are quite similar to path expressions in e.g. the UNIX file system (directory tree).

  However, XPath expressions are actually much more powerful. One could imagine a future operating system that uses an XDM tree (or something similar) to replace its file system.
- For example,

  `/GRADES-DB/STUDENTS/STUDENT`

  is an XPath-expression that selects `STUDENT`-nodes that are children of (the) `STUDENTS` node that is a child of the `GRADES-DB` document element.

Introduction (4)

- Path expressions are used also in object-oriented languages for navigating in complex structures.

  E.g., in OQL. Again, they are much simpler than XPath. By the way, there a full stop "." is used instead of "/". The relational model does not need path expressions because of its simple (flat) structure.
- One can view XPath as a simple query language for XML.

  It does not have joins and aggregations, but it has quite powerful selections, and it has certian forms of semi-joins.
- XPath has not itself XML syntax.

  This is more compact. Furthermore, XPath is used in attributes.

Introduction (5)

- XPath 1.0 is a W3C Recommendation since 16 November 1999.

  It began with work on the XSL Pattern Language, and the "location paths" in drafts of the XPointer specification. XPath unified the two.
- XPath 2.0 was published as W3C Recommendation on 23 January 2007.

  The main change from XPath 1.0 is the stricter typing. In 1999, when XPath 1.0 was published, there was no XML Schema yet (work on XML Schema had just begun, XML Schema 1.0 was published in May 2001). XPath 2.0 uses XML Schema types. Furthermore, variable bindings and nested subqueries were added. XPath 2.0 has a compatibility mode that removes most (but not all) incompatibilities with XPath 1.0.

Software (1)

- One can write a simple XSLT stylesheet that shows the result of an XPath expression. Then any XSLT processor (e.g., in a web browser) can be used.

  How to do this is shown below. Also links to XSLT processors are given that are independent of a browser (might give better error checking).

- An XPath expression is already a simple XQuery query. Thus, an XQuery processor can be used.

  XQuery implementations are listed below (some with online demo).

- XLab: Online XPath experiments

  [http://www.zvon.org:9001/saxon/cgi-bin/XLab/XML/xlabIndex.html?stylesheetFile=XSLT/xlabIndex.xslt]

Software (2)

XQuery Implementations:

- Galax

  Open source, from some authors/editors of the XQuery Specification. [http://www.galaxquery.org/]

- X-HIVE

  Commercial XML-DBMS, Online demo evaluator. [http://support.x-hive.com/xquery/].

- AltovaXML

  The engine used in XMLSpy is free (contains validator: DTD/Schema, XSLT 1.0/2.0, XQuery). [http://www.altova.com/altovaxml.html]

Software (3)

XQuery Implementations, continued:

- Qizx/open (open source Java implementation)

  [http://www.axyana.com/qizxopen/] Online demonstration: [http://www.xmlmind.com:8080/xqdemo/xquery.html]

- Saxon (from Michael Kay)

  Michael Kay is editor of the XSLT 2.0 specification. The basic version of Saxon (without static type checking and XQuery→Java compiler) is open source. It includes support for XSLT 2.0, XPath 2.0 and XQuery 1.0. [http://saxon.sourceforge.net/]

- eXist (open source native XML database)

  [http://exist.sourceforge.net/]
  Online demo: [http://demo.exist-db.org/sandbox/sandbox.xql]

Software (4)

XSLT Implementations:

- Any modern web browser has XSLT support.

  See, e.g., `http://www.mozilla.org/projects/xslt/`.

- Xalan (Apache)

  [http://xalan.apache.org/]

- XT (James Clark)

  [http://www.blnz.com/xt/index.html], [http://www.jclark.com]

- Sablotron

  [http://www.gingerall.org/sablotron.html]

- See above: Saxon, AltovaXML.

Trying XPath with XSLT (1)

- Modern web browsers can apply an XSLT stylesheet to visualize XML (by transforming it to HTML).
- Thus, one writes a reference to the stylesheet in the XML data file (input for XPath query), e.g.:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
        href="query.xsl"?>
<GRADES-DB>
    ...
</GRADES-DB>
```


Trying XPath with XSLT (2)

- Then one looks at this data file in the browser. It automatically fetches the stylesheet `query.xsl` (see next four slides) and uses it for the transformation.
- The stylesheet file mainly contains a transformation rule that evaluates an XPath expression (with the root node as starting point) and only shows the result of this expression in the output.
- However, additional transformation rules are necessary to format the result of the XPath expression (arbitrary XDM nodes) as HTML.

Trying XPath with XSLT (3)

```
<?xml version="1.0"?>

<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

  <xsl:output method="html"
      encoding="ISO-8859-1"
      doctype-public="-//W3C//DTD HTML 3.2 Final//EN"
      indent="yes"/>
```


Trying XPath with XSLT (4)

```
<xsl:template match="/">
    <html>
    <head><title>Query Result</title></head>
    <body>
    <ul>
        <xsl:apply-templates
            select="//STUDENT/LAST"/>
            <!-- This is the XPath expression
                 to be tested -->
    </ul>
    </body>
    </html>
</xsl:template>
```

Trying XPath with XSLT (5)

```
<xsl:template match="*">
   <li>ELEMENT: <xsl:value-of select="name(.)"/>
               (<xsl:value-of select="."/>)</li>
</xsl:template>

<xsl:template match="@*">
   <li>ATTRIBUTE: <xsl:value-of select="name(.)"/>
               (<xsl:value-of select="."/>)</li>
</xsl:template>

<xsl:template match="text()">
    <li>TEXT: <xsl:value-of select="."/></li>
</xsl:template>
```


Trying XPath with XSLT (6)

```
<xsl:template match="comment()">
    <li>COMMENT: <xsl:value-of select="."/></li>
</xsl:template>

<xsl:template match="/">
   <li>DOCUMENT: <xsl:value-of select="."/></li>
</xsl:template>

<xsl:template match="processing-instruction()">
   <li>PROC-INSTR: <xsl:value-of select="name(.)"/>
               (<xsl:value-of select="."/>)</li>
</xsl:template>

</xsl:stylesheet>
```


Trying XPath with XSLT (7)

- Older versions of Internet Explorer (IE 5) implemented an XSLT working draft, which is not very compatible.
- An example stylesheet is shown on the next two pages. Note the different namespace prefix.
- But the XPath expressions were very restricted. Thus, it is not clear whether today it is still useful to learn XPath in this way.

  E.g. the long form of path steps is not understood, ".." is not understood. Also, node positions are counted from 0 instead of from 1.

Trying XPath with XSLT (8)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- For Internet Explorer 5 (old, incompatible) -->
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/TR/WD-xsl"
        version="1.0">
    <xsl:template match="/">
        <html>
            <head><title>Query Result</title></head>
            <body><ul>
                <xsl:apply-templates
                    select="//STUDENT/LAST"/>
            </ul></body>
        </html>
    </xsl:template>
```

Trying XPath with XSLT (9)

```
    <!-- Format nodes as HTML in IE5 -->
    <xsl:template match="@*|node()">
        <li><xsl:value-of select="."/></li>
    </xsl:template>

</xsl:stylesheet>
```


Overview

1. Introduction, Software
2. Location Paths
3. Expressions
4. Data Types
5. XPath Functions

Context (1)

- An expression is evaluated relative to a context.
- In XPath 1.0, the context consisted of:
  - a node (context node)
  - a context position (position of context node in current set/sequence: positive integer 1, 2, ...)
  - a context size (number of nodes in current set: positive integer)
  - a set of variable bindings
  - a function library
  - a set of namespace declarations

Context (2)

- XPath 2.0 distinguishes static and dynamic context of an expression.
- The reason is that XPath expressions can possibly be compiled and optimized, and afterwards executed many times on different documents.
- In this phase, also static type checking is done.
- The actual (dynamic) types of the values that are computed during evaluation of an expression are either equal to the static type of the expression or more specific (derived from the static type).

Context (3)

- Dynamic context:
  - ◇ context item (atomic value or node)
  - ◇ context position
  - ◇ context size
  - ◇ variable values
  - ◇ function implementations
  - ◇ current dateTime
  - ◇ implicit timezone
  - ◇ available documents
  - ◇ available collections, default collection

Context (4)

- Remarks about dynamic context:
  - ◇ If the context item is a node, it is called context node.
  - ◇ Context item, context position and context size are together called the focus of an expression.
  - ◇ The current dateTime is used for the XPath function `current-dateTime`.

    It is guaranteed that if this function is accessed multiple times during an evaluation of an expression, it always returns the same value. This simplifies optimizations.

Context (5)

- Remarks about dynamic context, continued:
  - ◇ The implicit timezone is used for `dateTime`-values without timezone (“local time”) when comparing them with values with timezone (UTC).

    This seems not quite compatible with the XML Schema specification which treats values in local time as if they could possibly be in any timezone, leading to a partial order.
  - ◇ Available documents and collections are used for the functions `doc` and `collection`.

    `doc` maps a URI to a document node, and `collection` maps a URI to a sequence of nodes. The function `collection` can also be called without argument, then it returns the default collection.

Context (6)

- An important part of the static context is type information.
- XPath is always used embedded in another language (e.g. XSLT, XQuery).
- There are many parameters that are needed for evaluating an XPath expression that must somehow be set in the host language (e.g., namespaces).

  Also collations are needed for string comparisons.
- These are also part of the static context.

Context (7)

- Static context:
  - XPath 1.0 compatibility mode.

    This is true when the XSLT version is not 2.0.
  - Statically known namespaces.

    I.e. the namespace prefixes declared for the XPath expression.
  - Default namespace for element and data types.

    In XSLT, this can be set with `xsl:xpath-default-namespace="URI"`.
  - Default namespace for functions.

    XPath functions are in `http://www.w3.org/2005/xpath-functions`. XSLT automatically initializes this component of the static context with the standard namespace, so no prefix is needed when calling XPath functions.

Context (8)

- Static context, continued:
  - Schema information (types/elements/attributes)
  - Variable declarations (name and type).
  - Static type of context item.
  - Function signatures (name, input/result types)
  - Known collations, default collation.
  - Base URI.
  - Statically known documents/collections.

    The default type for a call to `document` is `document-node()?`, and for `collection`, it is `node()*`. If information should be available already during compilation, the types could be different (more specific?).

Location Paths (1)

- The purpose of an location path (or "path expression") is to select nodes in an XDM tree.

  Actually, in its very last step, it can also compute a sequence of atomic values (or a single value), not only a sequence of nodes.
  A path expression is not the most general kind of XPath expression, but it is the kind that is most often used.
- There are absolute and relative paths:
  - An absolute path starts with a "/" or "//", followed by a relative path.

    For "/", the relative path is optional. For "//", it is required.
  - A relative path consists of a series of steps, separated by "/" or "//".

Location Paths (2)

- The "//" will later be explained as an abbreviation:
  - The syntax must be defined including all abbreviations.
  - For the semantics, it suffices to treat only XPath expressions, in which the abbreviations are fully expanded (normalized expressions that do not contain e.g. "//").
- A step can be
  - an axis step (in full or abbreviated syntax),
  - a filter expression.

Location Paths (3)

- An axis step in full (verbose) syntax has the form

  axis::node-test[predictate]

  The predicate may be missing or may be repeated.
- The axis (e.g., `child`) selects a sequence of nodes by their position relative to the context node.
- The node test selects a subset of these nodes by their name or type (kind).
- The predicate(s) contain further conditions on the resulting nodes (e.g., position, value).

Location Paths (4)

- A filter expression consists of a primary expression followed by a sequence of zero or more predicates in “`[...]`”.
- A primary expression is:
  - ◇ Any XPath expression in parentheses `(...)`.
  - ◇ A data type literal (constant), e.g. `"abc"`.
  - ◇ A function call.
  - ◇ A variable reference, e.g. `$x`.
  - ◇ A context item reference: “.”

Location Paths (5)

- `E1/E2` is evaluated as follows:
  - ◇ `E1` is evaluated. The result must be a (possibly empty) sequence of nodes, otherwise a type error is raised.
  - ◇ `E2` is evaluated once for every node in the result of `E1` as context node.

    The context size is the length of the result of `E1`. The context position is the position of the context node in the sequence (depending on the axis, the position might be counted from the end of the sequence, see below).

Location Paths (6)

- Evaluation of `E1/E2`, continued:
  - ◇ If each evaluation of `E2` returns a sequence of nodes, the result of `E1/E2` is the union of the nodes in these sequences in document order (with duplicates removed).
  - ◇ If each evaluation of `E2` returns a sequence of atomic values, these sequences are concatenated (without duplicate elimination).
  - ◇ If `E2` returns nodes and atomic values, a type error is raised.

XPath Axis (1)

- An axis selects a sequence of nodes based on their position in the document tree relative to the current context node.
- There are 13 axis (in XPath 1.0 and in XPath 2.0).
- Of these, 8 are forward axes (cont. on next page):
  - ◇ `self`
  - ◇ `child`
  - ◇ `descendant`
  - ◇ `descendant-or-self`
  - ◇ `following-sibling`

XPath Axis (2)

- Forward axes, continued:
  - ◇ `following`
  - ◇ `attribute`
  - ◇ `namespace` (deprecated, not in XQuery)
- There are 5 reverse axes:
  - ◇ `parent`
  - ◇ `ancestor`
  - ◇ `ancestor-or-self`
  - ◇ `preceding-sibling`
  - ◇ `preceding`

XPath Axis (3)

- A minimal XPath implementation needs to support only the following axes:
  - ◇ `self`
  - ◇ `child`
  - ◇ `parent`
  - ◇ `descendant`
  - ◇ `descendant-or-self`
  - ◇ `attribute`

XPath Axis (4)

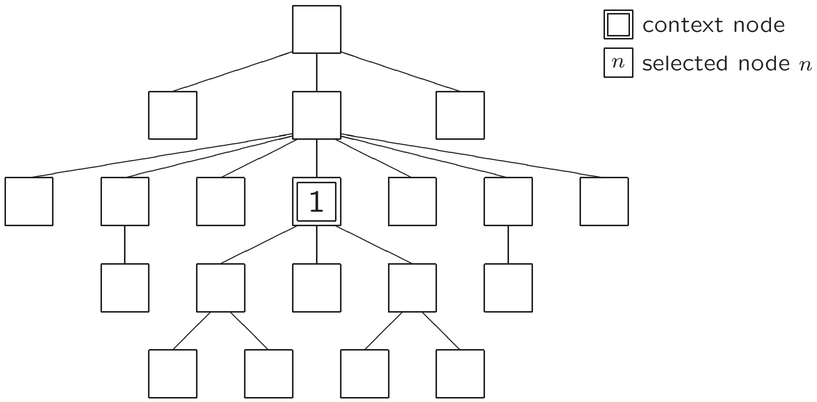
- The following axes partition a document (except attribute and namespace nodes): `self`, `ancestor`, `descendant`, `preceding`, `following`.
- If an axis is a reverse axis, the context position used for evaluating predicates in this location step is assigned in inverse document order.

  For forward axes, it is assigned in document order. If the predicate is not in a location step, the position is the position in the sequence.
- The selected nodes with their position are shown in an example on the following slides.

  The context node is marked with a double border.

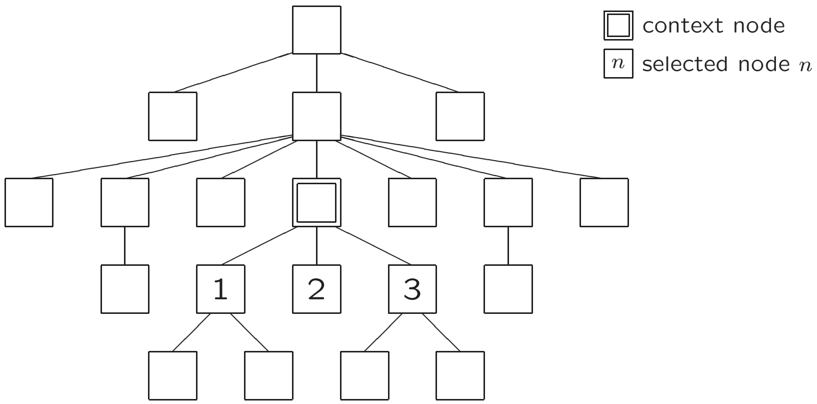
XPath Axis (5)

self:



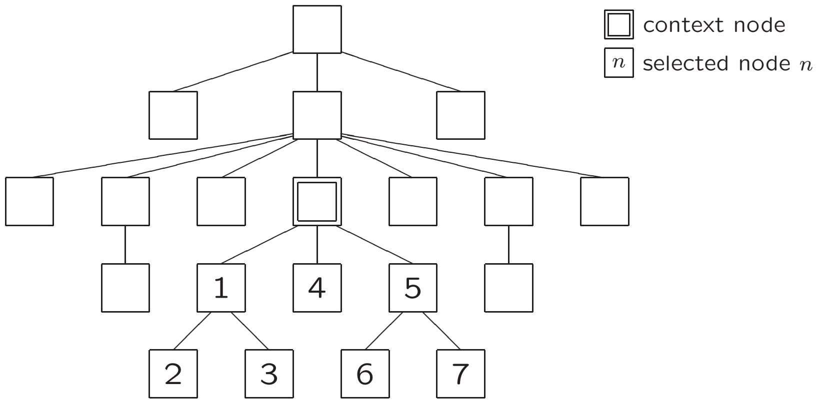
XPath Axis (6)

child:



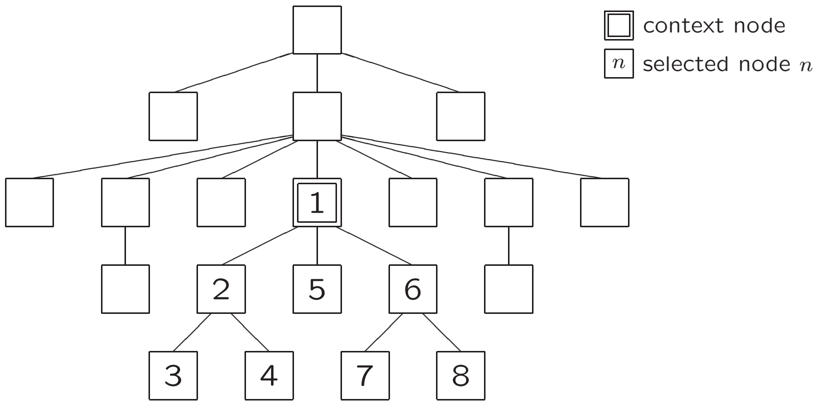
XPath Axis (7)

descendant:



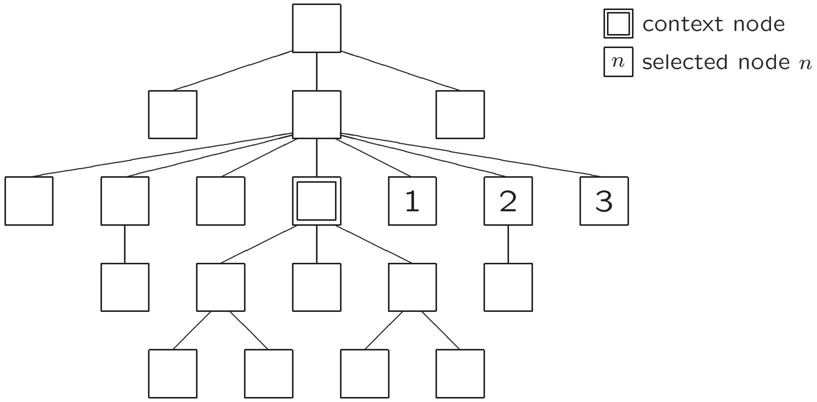
XPath Axis (8)

descendant-or-self:



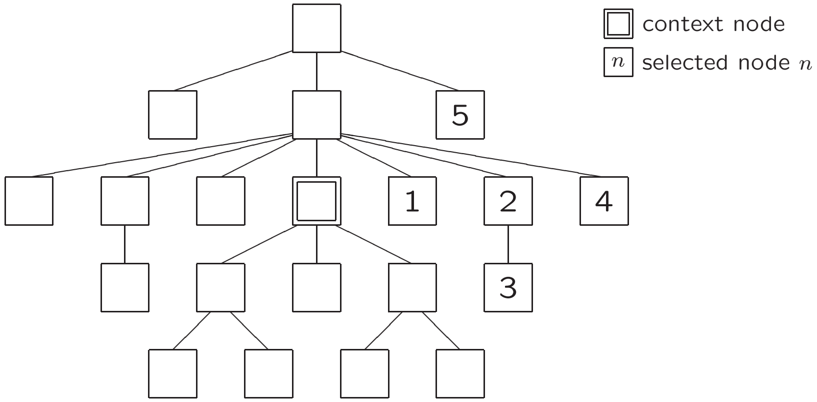
XPath Axis (9)

`following-sibling`:



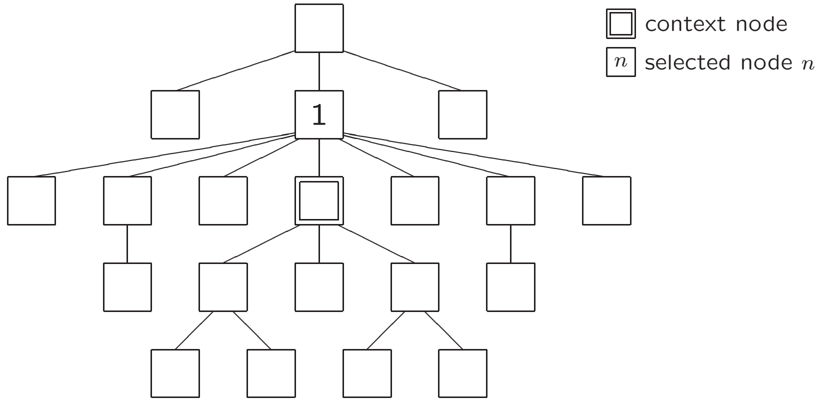
XPath Axis (10)

`following`:



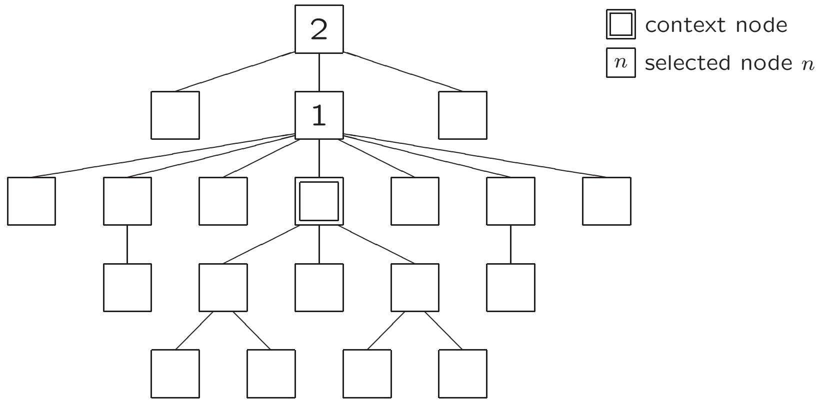
XPath Axis (11)

`parent`:



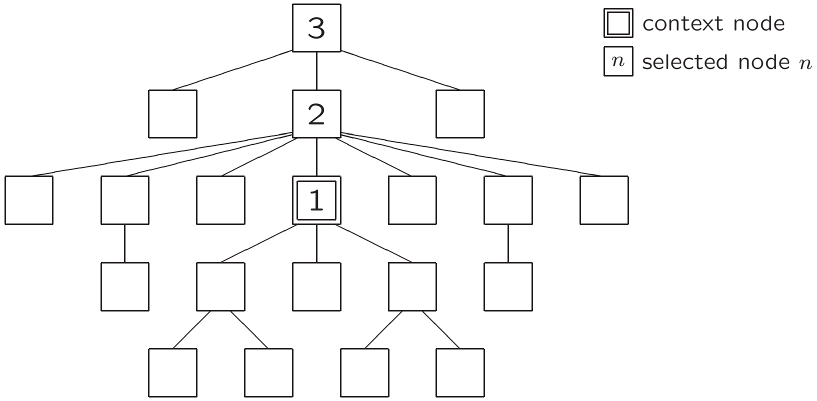
XPath Axis (12)

`ancestor`:



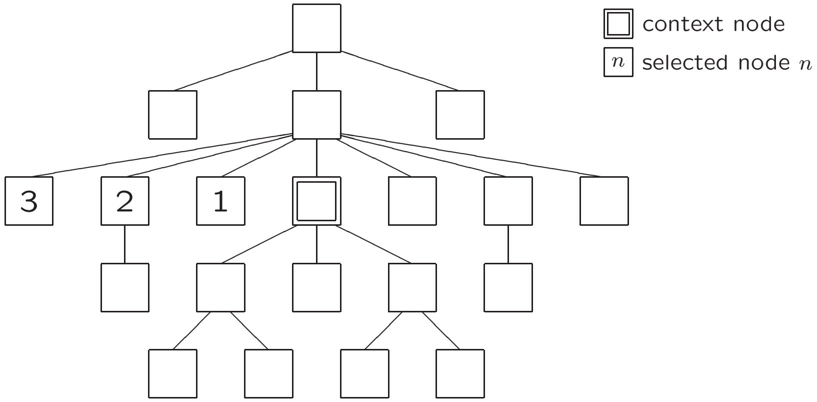
XPath Axis (13)

ancestor-or-self:



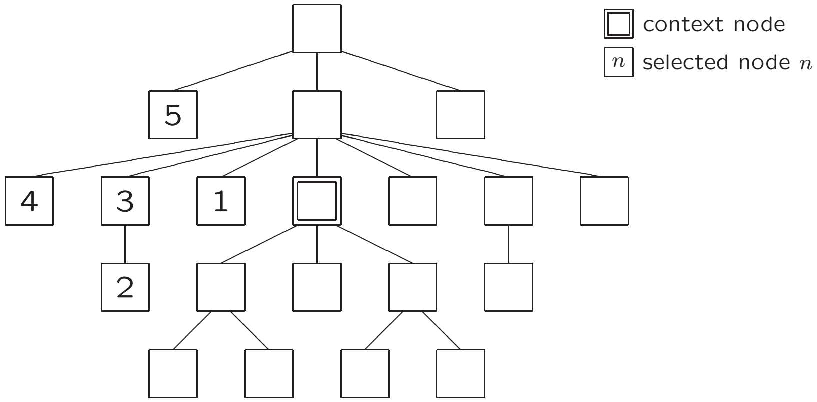
XPath Axis (14)

preceding-sibling:



XPath Axis (15)

preceding:



Node Tests (1)

- A node test is a name test or a node type test.
- In XPath 1.0, a name test had one of the forms
  - ◇ `QName` (local name or prefix:local name)

    Note that the standard default namespace declaration does not apply to XPath. Furthermore note that the namespace URIs are compared, not the prefix.
  - ◇ `NCName:*` (arbitrary name in given namespace)
  - ◇ `*` (no restriction)
- If a name test is used, the node type must be the principal type of the axis, which is "element" for all axis except the attribute and the namespace axis.

Node Tests (2)

- In XPath 1.0, the node types that could be used as node tests were:
  - ◇ `comment()`
  - ◇ `text()`
  - ◇ `processing-instruction()`
  - ◇ `processing-instruction('target')`
  - ◇ `node()`: All nodes reachable by the given axis.

    There, the node type is e.g. "`comment`", and the "()" makes it a node test. The problem is that there could be an element type "`comment`", and the "()" distinguishes the node type test from the name test.

    There were no node type tests for attribute and namespace nodes, because they are accessed via specific axis, and for document nodes, because this is accessed via "/".

Node Tests (3)

- In XPath 2.0, sequence type syntax was introduced. It defines a notation (name) for sequence types.
- Possible sequence types are:
  - ◇ `empty-sequence()`
  - ◇ A node kind test (see below), optionally followed by an occurrence indicator (`?`, `*`, or `+`)
  - ◇ `item()` with an optional occurrence indicator

    Remember that an item is a node or an atomic value.
  - ◇ an atomic type name (e.g., `xs:integer`) with an optional occurrence indicator.

Node Tests (4)

- The node kind tests in XPath 2.0 are:
  - ◇ `element(*)`: any element node

    This matches any element node. In an example, also `element()` is used, but the formal grammar does not seem to allow this.
  - ◇ `element(Name)`

    This matches any element node with the given name (QName).
  - ◇ `element(Name, Type)`, `element(*, Type)`

    This matches an element node with the given name (or any name in case of `*`) that is annotated with the type `Type`, or with a type derived from `Type`. The type can be followed by `?`, which permits nilled nodes. Otherwise nilled nodes would not match.

Node Tests (5)

- Node kind tests in XPath 2.0, continued:
  - ◇ `schema-element(Name)`

    This matches an element called `Name` or declared in a substitution group below `Name`. In addition, it must have the data type declared in the schema for the `Name`, or a more specific type. It can possibly be nilled if the element is declared as nillable. Basically, there must be a top-level declaration for `Name` in the schema, because the names of locally declared element types are implementation-dependent.
  - ◇ `attribute(*)`
  - ◇ `attribute(Name)`
  - ◇ `attribute(Name, Type)`, `attribute(*, Type)`
  - ◇ `schema-attribute(Name)`

Node Tests (6)

- Node kind tests in XPath 2.0, continued:
  - ◇ `document-node()`

    One can also use e.g. `document-node(element(GRADES-DB))`, and the same with the other forms of `element` and `schema-element` tests. The element test refers to the unique child element (document element). If there should be several child elements, the test fails.
  - ◇ `processing-instruction(Name)`

    The name can be a QName or (for backward compatibility) also a string.
  - ◇ `comment()`
  - ◇ `text()`
  - ◇ `node()`

Node Tests (7)

- The node name tests in XPath 2.0 are as shown above for XPath 1.0, only the new wildcard `*:...` was added (given local name, arbitrary namespace):
  - ◇ `QName` (i.e. `NCName` or `NCName:NCName`).
  - ◇ `*`
  - ◇ `NCName:*`
  - ◇ `*:NCName` (new in XPath 2.0)

Predicates (1)

- A predicate `[...]` filters an input sequence.
- It checks a condition for each item in the input sequence and yields an output sequence that contains only those items for which this condition is true.
- For each item in the input sequence, an "inner focus" is computed, i.e. the evaluation context is changed. With this context, the expression in `[...]` is evaluated.
- Once this is finished, one returns to the original context, i.e. the "outer context" (like a stack).

Predicates (2)

Evaluation of `E1[E2]`:

- `E1` is evaluated, let the result be sequence s.
- For each item x in s, an inner focus is computed as follows: The context item is x, the context size is the length of s, and the context position is basically the position of x in s.

  More precisely: If this predicate appears in a forward axis step, the context position is the position x would have if s were sorted in document order. If the predicate appears in a reverse axis step, the context position is the position x has between the nodes in s in inverse document order. If the predicate is not in a step, the context position is the position of x in s.

Predicates (3)

Evaluation of E1[E2], continued:

- For each x in s (result of evaluating E1), E2 is evaluated in the focus described above.
  - ◇ If the result is a numeric value, it is compared with the context position in this inner focus. If they are equal, x is appended to the output sequence.
  - ◇ Otherwise, the effective boolean value of the result is computed (see next slide). If it is true, x is appended to the output sequence.

Effective Boolean Value (1)

- Effective boolean value of an expression that returns value x:
  - ◇ If x is the empty sequence, the result is false.
  - ◇ If x is a sequence, the first item of which is a node, the result is true.
  - ◇ If x is a value of type `boolean` (or derived from `boolean`), the result is x.

    Formally, x is a singletom sequence containing a boolean, but singleton sequences are identified with the item they contain.
  - ◇ ... (continued on next slide)

Effective Boolean Value (2)

- Effective boolean value of x, continued:
  - ◇ If x is a `string` (or `anyURI`, `untypedAtomic` or derived from one of these), the result is false if it is the empty string, true otherwise.
  - ◇ If x belongs to a numeric type, the result is false if it is equal to 0 or `NaN`, true otherwise.
  - ◇ In all other cases, a type error is raised.
- Formally, this very generous conversion to `boolean` is done by the function `boolean(`x`)`.
- In many contexts, it is called implicitly.

Subtle Differences I

- Suppose that `STUDENT` has an attribute `GUEST` of type `boolean`. Then `[attribute::GUEST]` will be true when there is a `GUEST` attribute node, even if its value is false.

  One must explicitly take the value of the attribute with the `data(...)` function. Otherwise it checks only that the attribute node exists (which might be automatically inserted by applying a default value).
- The effective boolean value of `"false"` (a string) is true.

  `boolean("false")` is true, but `xs:boolean("false")` is false.

Abbreviated Syntax

- attribute:: can be abbreviated to "@".
- If no axis is given, the default axis is
  - ◇ "child::", unless the node test of that step is "attribute(...)" or "schema-attribute(...)".
  - ◇ In that case, the default axis is "attribute::".
- "//" is replaced by "/descendant-or-self::node()/".

  However, this may only be applied to a path expression that consists of something else besides "//". "//" by itself is not a legal path expression. In contrast, "/" is allowed.
- The step ".." is short for "parent::node()".

Meaning of Absolute Paths

- An absolute path can be understood as a relative path with first step

  root(self::node()) treat as document-node()
- Thus, it determines the root of the tree in which the context node is.

  E.g., by following the parent-link.
- This root node must be a document node, otherwise a runtime error occurs.

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 is the full version of the following expression?

  /*//AUTHOR/@LAST
- Please write an XPath expression for:
  - ◇ Print the last names of all authors.

    Assume that the context node is the document node and that it suffices to select the attribute nodes, and not necessarily take their value. E.g. <xsl:value-of select="..." separator=","/> would automatically take the value of the attribute nodes.
- What is the difference between the XPath expressions //TITLE and //TITLE/text()?

Subtle Differences II

- Note the difference between:
  - `//A[1]`: This selects all `A`-elements that are the first `A`-child of their parent.

    `//A[1]` stands for `/descendant-or-self::node()/child::A[1]`. Thus, `1` is the position for a `child`-step.
  - `/descendant::A[1]`: This selects only the first `A`-element in the entire document.
- Note also that these are not the same:
  - `//A[1]`: (as above, possibly many elements).
  - `(//A)[1]`: Only first `A`-element in document.

    Here, `[1]` applies to the entire sequence returned by `//A`.

Overview

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="'"'''"` 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.

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 $n \leq m$, 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 `|`.

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 comp. 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 "<".

Comparison Operators (2)

- 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)

- 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)

- 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 (`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 the all 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.

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_i.

    Unless the type of y_j is numeric, then `double` is chosen for x_i. E.g. if x_i is the string "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` if the general operator was `=`).

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.

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 (see Slide 6-145): 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.

Logical Conditions (1)

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

  The effective boolean value of the operands is automatically determined (see Slide 6-62). 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 6-122).

- `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 Slide 6-62).

- `=`, `<`, ... 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)

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

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

Exercise (2)

```
<?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 (3)

- 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]
```


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

  if(C) **then** E_1 **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)`

Overview

1. Introduction, Software
2. Location Paths
3. Expressions
4. Data Types
5. XPath Functions

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.

Overview

General Remarks (1)

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

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

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

General Remarks (2)

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

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

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

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

General Remarks (3)

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

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

General Remarks (4)

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

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

General Remarks (5)

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

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

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

General Remarks (6)

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

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

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

Subtle Differences III

- Let the context node be

  `<E A="3"/>`

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

- Then `1 to @A` works.

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

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

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

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

Node Properties (1)

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

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

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

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

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

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

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

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

Node Properties (2)

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

  Argument: `item()?`. Result: `xs:string`. Atomic values are casted to string, nodes are mapped to their string value (see Chapter 5, e.g. for element nodes, this is the concatenation of all decendant text nodes).

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

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

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

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

Node Properties (3)

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

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

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

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

Node Properties (4)

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

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

Finding Nodes (1)

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

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

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

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

Finding Nodes (2)

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

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

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

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

Finding Nodes (3)

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

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

Finding Nodes (4)

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

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

Functions for Sequences (1)

- s_1`, `s_2: Sequence concatenation (see Slide 6-81).

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

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

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

Functions for Sequences (2)

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

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

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

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

Functions for Sequences (3)

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

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

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

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

Functions for Sequences (4)

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

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

Exercise

- Suppose that the context node is

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

  Attribute a is declared as `xs:NMTOKENS`.

- What is the result of

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


Optimizer Hint

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

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

Aggregation Functions (1)

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

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

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

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

Aggregation Functions (2)

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

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

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

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

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

Exercise

- Consider again:

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

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

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


Boolean Functions (1)

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

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

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

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

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

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

Boolean Functions (2)

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

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

Boolean Functions (3)

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

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

    ```
    <E A="3" B="xyz"/>
    <E B="xyz" A="3.0"><!-- comment --></E>
    ```
  - ◇ Whitespace-only text nodes are not ignored in the comparison.

Numeric Functions (1)

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

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

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

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

Numeric Functions (2)

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

  The four numeric types are supported (see above). The result type is the same as the argument type. E.g. `round(1.2)=1.0`, `round(1.8)=2.0`. If x ends in `.5`, it is rounded upwards: `round(1.5)=2.0`, `round(-1.5)=-1.0`.
- `round-half-to-even(`x`, `$[n]$`)`: Round x to n decimal places to the right of the decimal point.

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

String Functions (1)

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

  Argument: `xs:integer*`. Result: `xs:string`. New in XPath 2.0.
- `string-to-codepoints(`s`)`: Map given string into sequence of Unicode character codes.

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

String Functions (2)

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

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

String Functions (3)

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

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

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

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

String Functions (4)

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

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

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

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

String Functions (5)

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

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

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

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

String Functions (6)

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

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

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

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

String Functions (7)

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

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

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

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

String Functions (8)

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

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

String Functions (9)

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

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

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

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

String Functions (10)

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

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

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

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

Regular Expressions (1)

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

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

Regular Expressions (2)

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

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

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

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

Exercise

- Consider again:

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

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

Context Functions (1)

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

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

- `position()`: Context position.

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

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

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

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

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

Context Functions (2)

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

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

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

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

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

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

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

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

URI Utility Functions (1)

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

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

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

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

URI Utility Functions (2)

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

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

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

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

- `iri-to-uri(`u`)`: Internationalized URI (IRI) → URI.

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

Namespaces

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

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

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

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

QNames

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

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

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

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

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

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

Error and Trace Functions

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

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

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

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