XPath: The Secret to Success with XSLT, XQuery, and Schematron

Debbie Lapeyre
Mulberry Technologies, Inc.
17 West Jefferson St.
Suite 207
Rockville MD 20850
Phone: 301/315-9631
Fax: 301/315-8285
dalapeyre@mulberrytech.com
`http://www.mulberrytech.com`

Version 1.0 (October 2017)

Mulberry
Technologies, Inc.

XPath: The Secret to Success with XSLT, XQuery, and Schematron

---

Appendixes

XPath: The Secret to Success with XSLT, XQuery, and Schematron

I. Objectives of the Course

- XPath 1.0 data model (thorough understanding)
- Location Paths (thorough understanding)
- Long and Short XPath Syntaxes (familiarity)
- XPath 2.0 and higher Data Model (exposure)
- Functions and Operators (exposure)

Almost all of XPath 1.0, some 2.0, mention of 3.0, three words on 3.1

This is *Not* New Technology

- XPath 1.0 1999 (used by programming languages)
- XPath 2.0 2007 (better! a weak programming language)
- XPath 3.0, 2014 (Turing complete programming language, supports streaming, higher order functions)
- XPath 3.1, 2017 (JSON-like maps and arrays)

(You need the fundamentals of XPath 1.0/2.0 before you learn XPath 3.0 [maybe])

A Few Examples of XPath

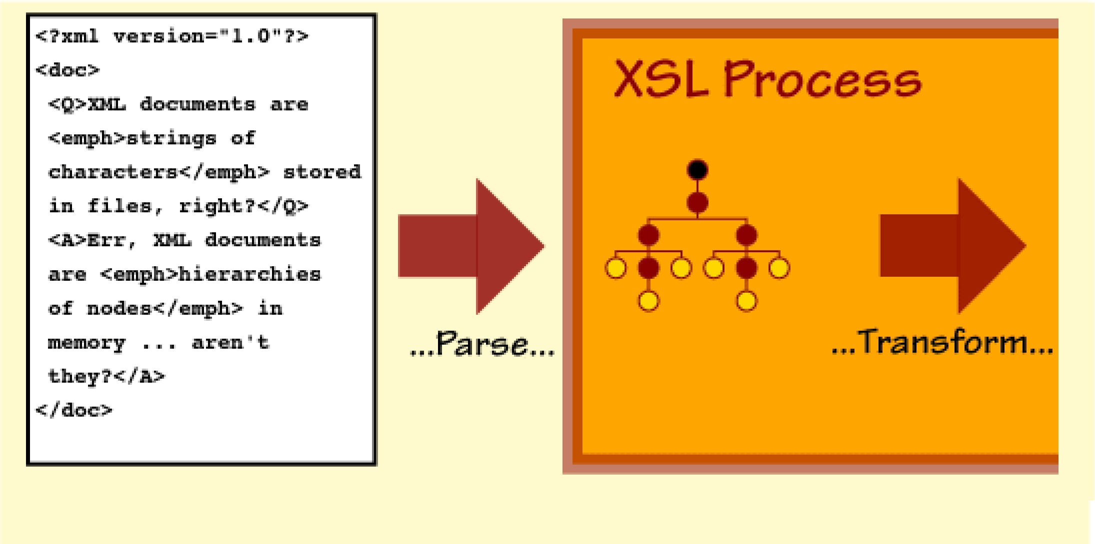
We are going to learn to read these

`//title`	Returns all `titles` in the document
`p[1]`	Returns the first `p` element child of the context node*
`attribute::security`	The “`security`” attribute on the context node*
`//div[@type='chapter']/figure`	Returns all `figure` elements inside `div` elements that have `type` attribute equal “`chapter`”
`child::book/child::title[contains(.,"XPath")]`	`title` children of the `book` children of the context node*, where the `title` contains the string “`XPath`”
`sum(child::cost)`	The sum of all the `cost` children of the context node*

(* “Context node” - wherever we are at the moment the XPath is evaluated)

II. Looking at an XML Document

- An XML document is a sequence of characters
    - data characters and markup characters
    - start-tag and end-tag markup delimits elements
- There is another way to think of an XML document (a tree!)
- Part of the processing (usually an XML parser) builds a tree
- Processes (like XPath and XSLT) work on trees of nodes (made from XML documents)

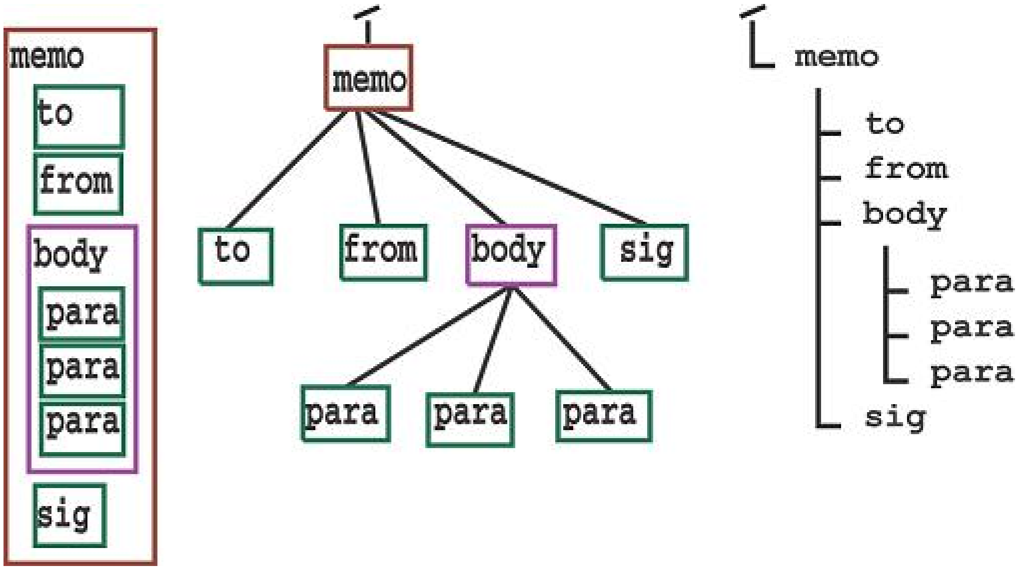


```
<?xml version="1.0"?>
<doc>
 <Q>XML documents are
 <emph>strings of
 characters</emph> stored
 in files, right?</Q>
 <A>Err, XML documents
 are <emph>hierarchies
 of nodes</emph> in
 memory ... aren't
 they?</A>
</doc>
```

(Text nodes were left out of this diagram to make it simpler to understand)

Getting from an XML Document to a Tree

- One ***root*** above all elements (called 'document node' or '/')
- Tree contains element nodes, attribute nodes, text nodes, etc.
- One ***document element*** (child of the root)
- “Containment” in XML becomes “children” in the tree



(Text nodes were left out of this diagram to make it simpler to understand)

Seven Types of Nodes in the Tree

- Root node (the one and only, “/”, aka “Document Node”)
- Element nodes (topmost one called “document element”)
- Attribute nodes
- Text nodes
  - For data character content of the elements
  - Includes whitespace-only nodes (usually line breaks)
- Comment nodes
- Processing Instruction nodes
- Namespace nodes (in XPath 1.0)

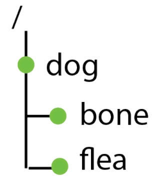
Note: The “document node” is *not* the same as the “document element”. Rather, the document element is a child of the document node (root).

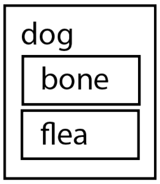
Nodes Have Name and/or Value Properties

- Some nodes have names (element nodes, attribute nodes)
- Each node has a *string* value
- Root node has
  - a name (/)
  - a value: the concatenation of all text nodes inside the whole document
- Element nodes have
  - a name (the "gi" or tag name)
  - a value: the concatenation of all text nodes inside the element
  - (document element value is a concatenation of all text nodes in the document)
- Attribute nodes have
  - a name (the name of the attribute)
  - a value (the value of the attribute)
- Text nodes have no names, just their text value

Tree Terms: parent, child, sibling

```
<dog>
  <bone/>
  <flea/>
</dog>
```

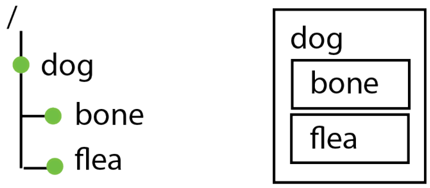




- ***/*** (root) is the *parent* of ***dog***
- ***dog*** is the *parent* of
  - ***bone***
  - ***flea***
- ***bone*** and ***flea*** are *children* of ***dog***
- ***bone*** and ***flea*** are *siblings* (of each other)
  - ***bone*** is a *preceding* sibling of ***flea***
  - ***flea*** is a *following* sibling of ***bone***

Tree Terms: Document Order

```
<dog>
 <bone/>
 <flea/>
</dog>
```



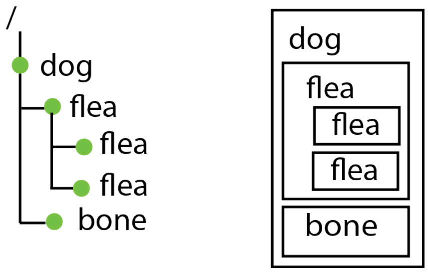
/
dog
bone
flea

dog
bone
flea

- Elements have a defined *document order*:
  1. `/`
  2. `dog`
  3. `bone`
  4. `flea`
- “*Depth-first* traversal”:
  means all the way down each branch before going on to next sibling

Tree Terms: ancestors, descendants

```
<dog>
 <flea>
  <flea/>
  <flea/>
 </flea>
 <bone/>
</dog>
```



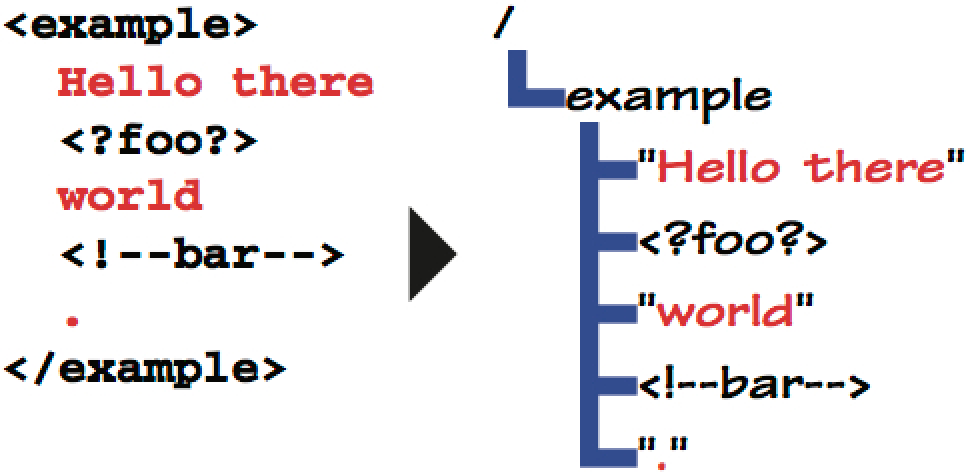
- Element ***dog*** has 1 ancestor: root (`/`)
- First ***flea*** has 2 ancestors: ***dog*** and root (`/`)
- 2nd/3rd ***flea***s have 3 ancestors: ***flea***, ***dog***, and root `/`
- ***dog*** has 1 ***flea*** child and 3 ***flea*** descendants
- ***root*** has 1 ***dog*** child and 5 descendants
- ***bone*** has 2 ancestors: ***dog*** and root (`/`)
- First `dog` element is *`ancestor`* of all the other elements and is called the *document element*
- `bone` has no children; it is *empty* (as are two of the fleas)

(document order: root, dog, flea, flea, flea, bone)

Making a Tree of Nodes

`<example>Hello there<?foo?>world<!--bar-->.</example>`

(Contiguous characters are grouped into *one* text node)



Optional Exercise, Making a Tree

```
<?xml version="1.0" encoding="UTF-8"?>
<article
  xmlns:xlink="http://www.w3.org/1999/xlink"
  article-type="book-review">
<front>
 <journal-meta>
   <journal-id journal-id-type="nlm-ta">Philos Ethics Humani Med</journal-id>
   <journal-title-group>
    <journal-title>Philosophy, Ethics, and Humanities in
         Medicine</journal-title>
   </journal-title-group>
   <issn pub-type="epub">1747-5341</issn>
   <publisher>
     <publisher-name>BioMed Central</publisher-name>
     <publisher-loc>London</publisher-loc>
  </journal-meta>
</front>
</article>
```

III. What is XPath?

- A language for
  - navigating to parts of the XML tree
  - performing operations over data (including, but not limited to, trees)
  - matching conditions in a tree (a subset of XPath is designed for this)
- Used in XSLT, XQuery, Schematron, XSL-FO, for XML databases, etc.
  - XPath says how to get there (in your document)
  - XQuery, XSLT, Schematron, XPath 3, etc. say what to do when you get there

(XPath 2.0, 3.0, and 3.1 may also tell you what to do)

XPath = The XML Tree-walking Language

- Named because it uses a path notation with slashes like UNIX directories and URLs

```
invoice/customer-data/customer-name
article/body/sec/title
/dog/flea/flea
```

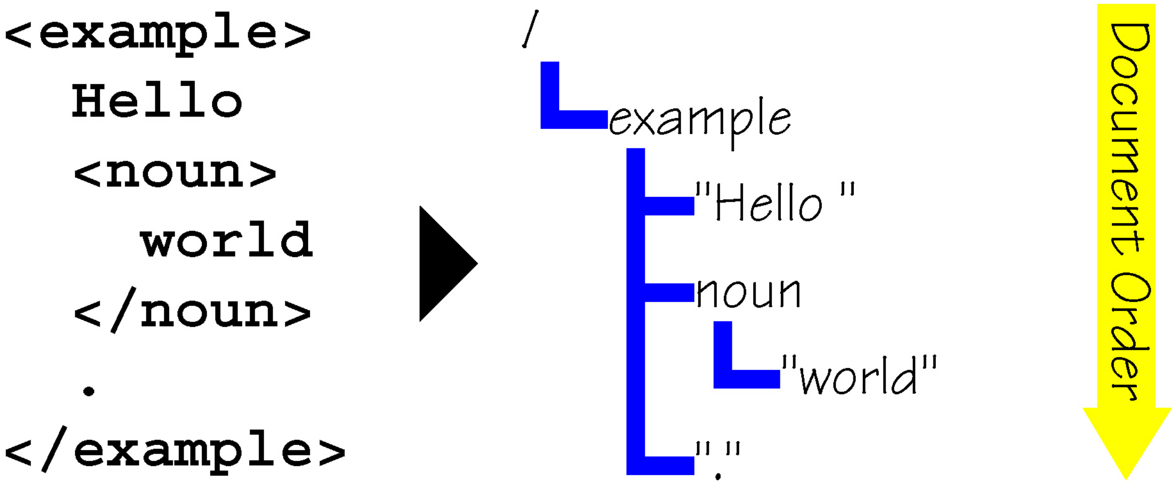
XPath has Three Main Uses

1. Locating portions of XML documents
   - addressing (naming) portions of an XML document
   - addresses (finds) a named portion of an XML document (“gimme my footnote!”)... and gets it back
2. Testing/Matching (used in Schematron, XSLT)
   - A subset of XPath was designed for this
   - Test whether a node in a tree matches a pattern
     (Is this node a paragraph inside a footnote with an attribute called “footnote-type” with value “legal”?)
3. Performing operations over data (including trees)
   - numeric operations (counting, adding, rounding)
   - string operations (contains, starts-with, substring, tokenizing)
   - boolean operations (for conditionals: equality, comparisons between numbers or nodes)
   - sorting, and lots more

The XPath 1.0 Data Model: Trees Not Text

- XPath does not
  - read or understand XML documents (tagged text)
  - understand about pointy brackets or entities
- XPath works on trees (a model of an XML document)
  - Some application makes an XML document into a *tree of nodes*
  - XPath works with element *nodes*, attribute *nodes*, comment *nodes*, etc.

An application uses XPath to select part of a tree for processing



```
<example>
  Hello
  <noun>
    world
  </noun>
  .
</example>
```

Axes: How XPath Talks About the Tree

- The parts of the tree are named using *axes* (for example, **`ancestor::`** or **`child::`**)
- An axis is a relationship between
  - “Where you are now” and
  - Another part of the tree
- “Where you are now” is called the *context node*
- An *axis* determines a direction to travel on the tree
  - Always starting from a *context node*
  - Always in one direction
  - This is one “step” in traversing the tree

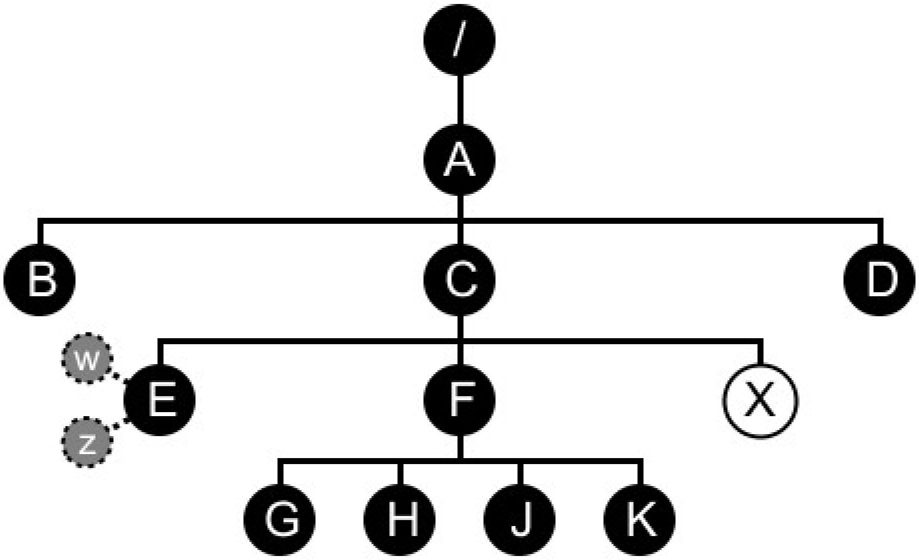
Syntax for Axes

- An XPath axis is written as
  - the axis name followed by
  - two colons
  - e.g., `parent::`
- “Forward” axes proceed in document order (like `child::`)
- “Reverse” axes proceed in reverse document order (like `ancestor::`)

The 13 XPath Axes

child	descendant	descendant-or-self
parent	ancestor	ancestor-or-self
attribute	following-sibling	following
self	preceding-sibling	preceding
	namespace	

Let’s Learn the Axes



(Text nodes were left out of this diagram to make it simpler to understand)

Optional Exercise: Gathering Nodes By an Axis

- Taking the node “X” as the context node
- Let’s run through the axes, one at a time

slide 22

Five Axes Cover All a Document’s Elements from Anyplace

The following five axes (taken together) let you cover the entire tree.

- ancestor(s) + descendant(s) + following + preceding + self = all nodes (except attribute and namespace)

slide 23

The Peculiarities of Attributes

- An attribute node has a parent (the element to which it is attached)
- But the attribute is not
  - a “child” of that parent
  - or a “descendant” either
- The only way to retrieve an attribute is to use
  - `attribute::` axis
  - short form `@`

```
article[attribute::status="draft"]
article[@status="draft"]
```

(The `child::` axis traverses to elements, text nodes, comments, or processing instructions, but not to attributes.)

XPath Location Paths Walk the Tree

(Deep, reread this after you see it!)

- Location Paths written as a series of “steps”
- Each step talks about nodes in the tree
- A slash (`/`) between each step
- Paths are composed left to right (beginning at the context node)
- Each step:
  - selects the requested nodes relative to the context node (selected in the previous step)
  - uses tests to determine which nodes to keep
  - Provides the context for the next step

`child::title[@xml:lang="en"]`

Each Location Step Has At Least Two Parts

(May Have Three)

1. ***Axis*** — Where to go (in relation to the context node)
    - expressed as an Axis name + “::” (`descendant::`)
    - an axis specifier is always present
    - sometimes implicit (`title` same as `child::title`)
2. ***Node Test*** — What kind of node do you want?
    - expressed as the name or type of the node
    - (`title`, `text()`), element()
3. ***Filter*** (also known as Predicate)
    - an optional qualifier to further refine/restrict the nodes returned
    - inside square brackets after the node test ([ ])
    - (`[starts-with(.,"The")]`, `[last()]`)

Location Step = `axis::` + `nodetest` + `[predicate/filter]*`

`child::title[@xml:lang="en"]`

A Step With Three Parts

`child::list[count(descendant::item) > 8]`

1. An ***axis*** (`child::`)
2. A ***node test*** (the name of an element “`list`”)
3. Zero or more ***predicates/filters*** `[count(descendant::item) > 8]`

Go along the child axis from the context node,
and gather up all the `<list>` elements,
then keep each `<list>`;
if and only if it has more than 8 `<item>` descendants.

Absolute Location Path

- Starts at the root
- Begins with a “/”
- `/body` retrieves all child elements *of the root* named `body`

```
/article/body
/article/body/section/title
/article/front/article-meta/pub-date
```

Relative Location Path

- Starts at the context node
- Has no leading “/”
- `body`
  - Starts *wherever we are at the moment*
  - Retrieves child elements *of the context node* named `body`

```
article/body
body/section/title
article-meta/pub-date
pub-date
```

“/” Separates Location Paths into Steps

Relative Location Path

`sec/title`

One or more “location steps” separated by “`/`”

Absolute Location Path

`/sec/title`

Initial “`/`” indicates the root node;
followed by a location path

Let’s Evaluate the Location Path `slide/title`

Two ways to read and use this Location Path:

- As a context or match pattern
  - matches any ***`title`*** child of a ***`slide`*** in the document
- As a select expression
  - starts at the context node
  - selects all ***`slide`*** children of the context node
  - then selects all the ***`title`*** children of those slides
  - returns a node list (union of ***`title`*** elements)
  - what is selected depends on the context node

Let’s Watch Select Expressions in Action

First, Determine the Context Node

Something non-XPath does this:

- Schematron @context attribute
- XSLT @match attribute

```
- <slideshow>
    <title>Introduction to XSLT for Managers</title>
  - <segment>
      <title>Overview</title>
    - <slide>
        <title>Administrivia</title>
      </slide>
    - <slide>
        <title>Where We Are Going Today</title>
      </slide>
    </segment>
  </slideshow>
```

The `slide` Step in `slide/title`

`slide/title`

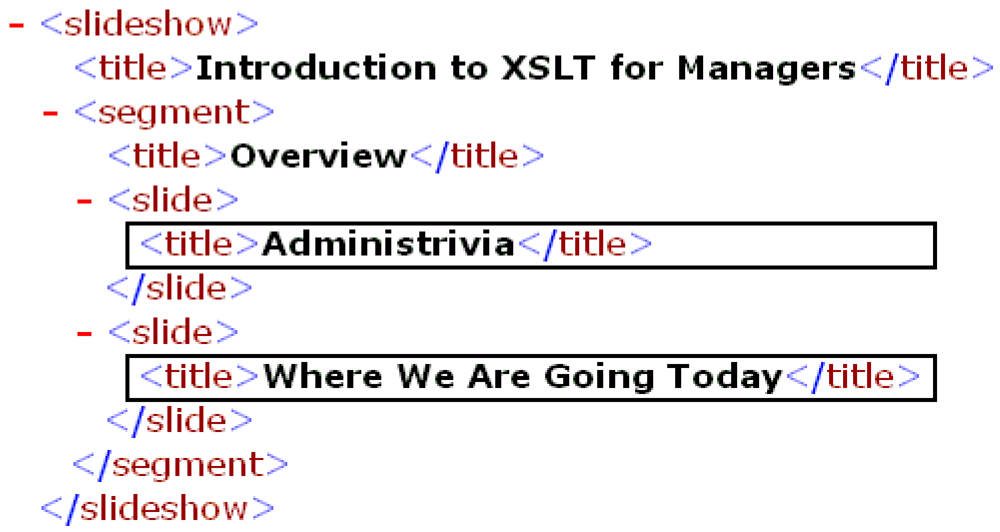
Select the ***`slide`*** children of context node:

```
- <slideshow>
    <title>Introduction to XSLT for Managers</title>
  - <segment>
      <title>Overview</title>
    - <slide>
        <title>Administrivia</title>
      </slide>
    - <slide>
        <title>Where We Are Going Today</title>
      </slide>
    </segment>
  </slideshow>
```

The `title` step in `slide/title`

`child::slide/child::title`

For each of those ***`slide`*** nodes, select ***`title`*** children:



```
- <slideshow>
    <title>Introduction to XSLT for Managers</title>
  - <segment>
      <title>Overview</title>
    - <slide>
        <title>Administrivia</title>
      </slide>
    - <slide>
        <title>Where We Are Going Today</title>
      </slide>
    </segment>
  </slideshow>
```

Result is the union

A More Complex Location Path (optional)

`slide[attribute::type="overview"]/list[count(descendant::item) > 8]`

- Still has two steps separated by “/” character:
- Step #1 `slide[attribute::type="overview"]`
- / (a slash)
- Step #2 `list[count(descendant::item) > 8]`

Stepping Through This Example (optional)

```
slide[attribute::type="overview"]/
list
[count(descendant::item) > 8]
```

- Step #1
  - From where we are (our context node)
  - Go through that node’s children
  - Get the `slide` elements
  - Take the ones that have a `type` attribute with the value “`overview`”
- Step #2, For each of the selected `slide` children
  - Get all its `list` children
  - Keep the ones that have more than eight `item` descendants

Homework: An Even More Complex Relative Location Path (optional)

```
/descendant-or-self::node()/child::body/descendant-or-self::node()/child::sec/
child::p/child::list/child::list-item[3]/child::p
```

(Explained on the next slide; try it first as a self-test.)

Stepping Through This Complex Example

(optional)

```
/descendant-or-self::node()/child::body/descendant-or-self::node()/child::sec/
child::p/child::list/child::list-item[3]/child::p
```

- Step #0, the root "/", this is an absolute path
- Step #1, all the descendants of the root, plus the root
- Step #2, all the `body` children of all these
- Step #3, all the descendants of the `body` element, plus the `body`
- Step #4, all the `sec` children of these elements. Yes, there are lots of them, not just the body's `sec` children but also *their* `sec` children
- Step #5, all paragraphs (`p` children) in each `sec`
- Step #6, all `list` children in each `p` element
- Step #7, the third `list-item` in each `list`
- Step #8, all the paragraphs (`p` children) in this `list-item`

XPath Node Tests

Location Step = `axis::` + **`node test`** + `[filter]*`

- Test the nodes in the tree
  - By type of node (element, comment, etc.)
  - By name of node (element type name (gi), attribute name)
- A common node test is “*”
  The meaning depends on the axis

`child::*`	means all element children of the context node
`attribute::*`	means all attributes of the context node

“*” selects all nodes of the “primary node type” of the axis

Node Testing by Name

- *`name`*
  - Tests the name of the node
  - Returns nodes of that name from the axis specified

`child::`**`item`**	Retrieves any child elements named `item`
`parent::`**`list`**	Retrieves a parent element named `list`
`attribute::`**`type`**	Retrieves any *attribute* named `type`
`ancestor-or-self::`**`section`**	Retrieves any ancestor elements named `section`, or the context node itself if it’s a `section` element

Node Testing by Type

You can use any of these node tests with any axis

`node()`	Test is `true` for any type of node
`text()`	Any text node
`comment()`	Any comment node
`processing-instruction()`	Any processing instruction node
`element()`	Any element node [XSLT 2.0+]
`attribute()`	An attribute [XSLT 2.0+]
`item()`	Any item (node or atomic value) [XSLT 2.0+]

Pop Quiz: `attribute::text()` gets you which nodes?

Node Testing by Explicit Schema Data Type

(XSLT 2.0+)

`element()`	Any element node
`element(title)`	Any element named `title` (any data type)
`element(title, hardtitle)`	Any element named `title` whose schema type is the user-defined type “`hardtitle`” (or a type derived from “`hardtitle`”)
`element(*, hardtitle)`	Any element whose schema type is the user-defined type “`hardtitle`” (or a type derived from “`hardtitle`”)
`element(*, xs:date)`	Any element whose schema type the simple type `xs:date`
`schema-element(title)`	Any element named `title` or in the substitution group headed by title and (loosely) whose schema type is the same as `title`’s (or a type derived from “`title`”)

For Reference: More Node Tests (optional)

- `processing-instruction(`*`$target`*`)`
  - Test is true for any processing-instruction node with target named `target`
  - `child::processing-instruction('xml-stylesheet')` retrieves any PI children with target named `xml-stylesheet`
- `$prefix:*`
  - True for any node of the principal node type of the axis in the namespace identified with the given prefix
  - `descendant-or-self::svg:*` retrieves any descendant elements in the `svg` namespace, or the context node itself if it is one
  - For example,

    `ancestor-or-self::`**`tei:div`**

    Retrieves any `ancestor` elements named `div` in the `tei` namespace, or the context node itself if it’s such a `div`

Expressions in Location Paths (optional)

A location step can include an expression

```
//mixed-citation/(name | person-group)/surname
```

If the expression is not the final step, it must return a sequence of nodes (or an error is returned)

Here is the same thing in XPath 1.0

```
//mixed-citation/name/surname  | //mixed-citation/person-group/surname
```

Filters (Predicates)

Location Step = `axis::` + `nodetest` + **`[filter/predicate]*`**

- A location path step
    - traverses the tree and
    - collects a set/list of nodes
- Each predicate *filters* that set of nodes
- Filters/Predicates appear within square brackets

A Sample Filter

`descendant::slide`**`[@showintoc='yes']`**

The XPath expression above retrieves

- Descendant elements of the context node named `slide`
- Then keeps only those that have
    - a `showintoc` attribute
    - with value equal to “`yes`”

Filters can be read as “if and only if” or “keep only those that”

Examples of Filters

`child::emph[@type]`

- `emph` element children with an attribute `type`

`child::emph[@type='italic']`

- `emph` element children with attribute `type` whose value is “italic”

`slide[descendant::title[contains(self::node(), 'Where We Are')]]`

- `slide` children
  - That have a descendant `title` element
  - That contains the string “`Where We Are`”

`contains()` is a function (two arguments)

One Step Can Take Many Filters

- Each successive predicate filters the node set to another node set
- Multiple predicates in a single step are evaluated *left to right*
  - Each predicate filters a node set
  - Each filtered node set provides the context for the next predicate (or next step if this is the last predicate)

Therefore order matters!!

`slide [@type] [3]`

- (`slide` children of context node, those with an attribute of `type`, the third such `slide`)

`slide [3] [@type]`

- (`slide` children of context node, the third such `slide`, *if and only if* that slide has an attribute of `type`)

Examples of XPath

`@security`	The “`security`” attribute on the context node
`sum(cost)`	The sum of all the `cost` children of the context node
`book/title[contains(.,"XPath")]`	`title` children of the `book` children of the context node, where the `title` contains the string “XPath”

`For $a in`
`distinct-values(/bib/book/author)`
`return ($a,`
`/bib/book[author = $a]/title`
	`For x in...`
	Returns a sequence of distinct values of `author` elements inside `book` elements, each author followed by the book `title` elements belonging to that author

Reading an XPath

Quiz: Figure out what you will get back

```
child::flea
```

```
ancestor::flea
```

```
//caption[count(*) > 1 or not(p)]
```

```
contrib-group/contrib
```

```
contrib-group[@content-type="author"]/contrib/(name | string-name)/surname
```

```
//sec[@type="summary"]
```

```
//sec[title | label]
```

```
//sec/title
```

```
//xref[@rid = current()/@id]
```

```
back/sec[@id and not(ancestor::appendix)]  |
sec/subsect1[@id and not(ancestor::appendix)] |
subsect1/subsect2[@id and not(ancestor::appendix)] |
subsect2/subsect3[@id and not(ancestor::appendix)]
```

All the rest is which ones, not what

XPath Short and Long Syntax

Long syntax:

- Explicit
- Easy to learn
- Can be verbose

Short syntax:

- Some long forms can be abbreviated
- Concise, easy to use (if you know what it means!)
- But there are a few “gotchas”
  some things don’t work with short, only with long

Heads-up: Long and Short Syntax

- XPath has an abbreviated (short) syntax for some constructions
  - `child::slide[attribute::type="overview"]`
    is the same as
    `slide[@type="overview"]`
- Most XPath in real life uses short syntax *when possible*
- Some things can only be expressed in long syntax
- Short syntax is fun and easy when you know long syntax
  ...and confusing (no fun!) when you don’t

So we learn the long syntax first

Abbreviations to Make Short Syntax

Full Syntax	Abbreviated Syntax	Comment
`child::`		no axis means the `child::` axis
`attribute::`	`@`	
`/descendant-or-self::node()/`	`//`	Note that this is one full step: axis, node test, and delimiting slashes
`self::node()`	`.`	i.e., the context node
`parent::node()`	`..`	
`[position() = 12]`	`[12]`	A number (or expression returning a number) by itself in a predicate is an equality test against `position()`

...and that's it!

Short Syntax Simplifies Expressions

`child::slideshow/ child::title`	`slideshow/title`
`parent::node()/ descendant-or-self::node()/ child:title`	`..//title`
`self::node()/ descendant-or-self::node()/ child::emph/ attribute::type[self::node()='italic']`	`.//emph/ @type[.='italic']`

Optional Exercise: Long and Short Syntax

- Take a look at the XPath 1.0 reference card
- Translate the expressions in the tables from full syntax to abbreviated syntax or from abbreviated to full.

From Full to Abbreviated

Translate the expressions from full to abbreviated syntax

Full Syntax	Abbreviated Syntax
`self::node()/child::PROLOGUE/child::TITLE`	
`/descendant-or-self::node()/child::STAGEDIR`	
`child::*/child::LINE`	
`parent::node()/child::processing-instruction("foo")`	
`attribute::bar`	

From Abbreviated to Full

Translate the expressions from abbreviated to full

Abbreviated Syntax	Full Syntax
`PERSONA`	
`./PGROUP`	
`//FM/P`	
`/`	
`SCENE/LINE`	
`../TITLE`	

(Answers are in Appendix A)

Warning: In a Location Path, Axis and Node Test Are Required

- Watch out! *Every step* has an axis and a node test.
- Abbreviations (short syntax) may make things invisible — but *they're still there*
- (Except filters. When they're not there, they're not there.)

This is good. It means when a location path is mysterious, all you have to do is expand it to long syntax and figure out what its pieces are.

IV. Match Patterns are A Subset of XPath

Remember there are two ways to read and use this Location Path:

`slide/head`

- For a select expression,what is selected depends on the context node
  - starts at the context node
  - selects all its ***`slide`*** children
  - then selects all the ***`title`*** children of those slides
  - returns a node list (union of ***`title`*** elements)
- As a context or match pattern
  - matches any ***`title`*** child of a ***`slide`*** in the document
  - used in Schematron `@context` attribute
  - used in XSLT `@match` attribute

The XPath of *Match Patterns*

- Are a *subset* of XPath expressions returning node sets
- Have special “match pattern” rules:
  - Only `child::` and `attribute::` axes are allowed
  - `/` and `//` step operations are allowed
  - Filters are allowed
  - XSLT 1.0 disallows variable references; XSLT 2.0, 3.0+ allow variables

A good match pattern	Not okay
`sec`	`following-sibling::*`
`caption/title`	`title/parent::caption`
`sec//p`	`sec/descendant::p`
`caption[title]`	`caption/title/..`
`p[1]`	`p[position() = $pos]`
`sec[@sec-type='chapter']/title`	`1 + 2`

Examples of Using Match Patterns

Ancestry

`<xsl:template match="title">`

vs

`<xsl:template match="sec/title">`

vs

`<xsl:template match="sec/sec/title">`

Associated Values

`<xsl:template match="ext-link[@ext-link-type='uri']">`

vs

`<xsl:template match="ext-link[@ext-link-type='email']">`

Arbitrary Criteria

`<xsl:template match="list-item">`

vs

`<xsl:template match="list-item[not(following-sibling::list-item)]">`

So What is the Problem?

- Match patterns and select expressions have the same syntax
- So they can look *just alike*
- Which can be confusing

A Plain Old Location Path

`<xsl:apply-templates select="`**`sec/title`**`"/>`

- As an XSLT select expression
  - Selects a set of nodes for processing
  - Evaluated relative to the current node
  - Returns a list of nodes (all the `title` children of the section (`sec`) children of the context node, in document order)

Same Location Path as a *Pattern*

In Schematron, we have:

`<rule `**`context="sec/title"`**`>`

Matches a node if and only if:

- Node is a `title`
- Node has `sec` parent

(Optional Exercise: Let’s all go see Appendix B for more about location paths versus patterns.)

V. XPath is an “Expression Language” (advanced)

When you write XPath, what you write is an expression

- A location path is one kind of expression
  `/article/front/article-meta/pub-date`
- `(7 * 6)` is also an expression
- An expression is evaluated to produce an object
  - A location path returns a sequence (list) of nodes
  - `(7 * 6)` returns `42`
  - `"XPath" = "difficult"` returns `false`
  - `distinct-values( (4,5,6,7,6,5,4) )` returns a sequence `(4,5,6,7)`

`distinct-values()` is a function “`(4,5,6,7,6,5,4)`” is a sequence

XPath Defines Functions and Operators (for Expressions)

- Syntax for a function is:
    - the name of the function followed by
    - parentheses, which contain
    - any arguments the function needs (maybe none!)
- For example
    - `count(item)` returns a count of the number of `item` children
    - `contains("Mulberry", "M")` returns `true` (boolean)
    - `not(title)` returns `true` if the context node has no `title` child and `false` if it has one (boolean)
    - `concat('Mu','lberry')` returns “`Mulberry`” (a string)
    - `starts-with('Mulberry', 'M')` returns `true` (boolean)
    - `distinct-values($someSequence)` returns a list of the non-duplicate items in the given sequence
    - `last()` returns a number equal to the context size

More Examples of Functions (optional)

- `item[`***`position()`***` = 3]`
  - Get `item` element children whose position is 3 (i.e. the third one)
- `item[`***`last()`***`]`
  - Get `item` element children whose position is equal to the number of `p` elements on the axis (i.e. the last one)
- `slide[`***`count(list)`***` > 1]/head`
  - Get `slide` element children that have more than one `list` element child; then get the list's `head` element children
- `child::*[`***`not(self::contrib)`***`]`
  - Get any element children that are not themselves `contrib` elements
- `//title[*]`
  - Just a filter, not a function. Get all the `title` elements that have children
- //***normalize-space(title[not(*)])***
  - Get rid of extra whitespace on all the title elements that DO NOT have any children
- `attribute::*[`***`not(local-name()='type')`***`]`
  - Get attributes that aren't named "`type`"

For Reference: Some Useful Functions

(optional)

`position()`	Returns position of node in current node list (*Not* the position of the node in the document)
`last()`	Returns the size (count) of the current node list
`count(`*`node-set`*`)`	Counts the nodes in the argument node set
`name(`*`node-set?`*`)` and `local-name(`*`node-set?`*`)`	Returns the namespace-qualified name (`name()`) or local name (`local-name()`) of a node (the first node in the argument node set)
`not(`*`object`*`)`	Converts the argument to a Boolean (when necessary) and inverts it

Optional Exercise: Looking at Some XPath Functions

We might look at Appendix C

All Functions, Expressions, Operators Work on Typed Data

- Best if types are explicit (from schema or casting)
- XSLT 1.0 will “coerce” type if there is no typing specified
- XSLT 2.0+ throw an error on type mis-match
- You can test on types as well as on elements and attributes

Most Common Types for Expressions

- Nodes of zero or more nodes
  (node sets in XSLT 1.0, sequences of nodes in XSLT 2.0+)
- Numbers (1, 2, 3, 30000000, `NaN`)
- Strings (“Debbie”, “Tommie”, “1”, “30000000”)
- Booleans (`true` or `false`)
- Sequences of “items”

(XSLT 2.0 and 3.0+ can also use all Schema types, derived types, and atomic values)

XPath 1.0 Assumes Automatic Casting Between Data Types

(There’s magic in those expressions!)

- Some functions/operations require an argument or operand of a particular type
- If the given arguments are not what the function needs
  ...will try to turn an object into what it needs

```
concat('http://', child::url)
```

- `concat()` requires strings as arguments
- The first argument is a ***string***; but the second is a *node set*
- `child::url` will be turned into a ***string***
  - A node set is converted into a *string* by taking the string value of the first node in the set (in document order)
  - The concatenation could produce “`http://www.mulberrytech.com`”
  - If there is no node, or an empty one, you’d get “`http://`”

XPath 1.0 Rules for Converting Objects to Booleans

- There are rules for converting between objects (Appendix C)
- Here are the rules for converting an object to a boolean
- First column is object you have; second column is how the conversion works

Boolean	`false` if false, `true` if true
Number	`false` if zero, `true` if not
String	`false` if empty (`= ""`), `true` if not (or true if the string’s length >= 1)
Node set	`false` if empty (no nodes in set), `true` if not

XPath 2.0 and XPath 3.0 Types are Explicit

New functions to deal with types

- Create types explicitly
- Cast between types
- Determine (or fix) types before you try to use them
- Catch type errors with conditional testing

Schema-aware processors (SA) can read the types from the schema

Type Functions (optional)

`cast as`	Creates a new value of a specific type based on the existing value *input-expression* **cast as** *target-type* `5 cast as integer`
`castable as`	Tests if a given value can be cast into a given target type without error. Returns boolean. *expression* **castable as** *target-type* `$size castable as xs:anyURI`
`xs:date("2017-10-31")`	Constructor functions. One for every one of the atomic XSD types. Requires xs: namespace. This one works the same as `("2017-10-31" cast as xs:date)`
`instance of`	Returns boolean if the value of the first operand matches the type given in the second operand `3 instance of xs:integer` would return “ `true` ”
`treat as`	At run time, here is the type you should have; postpone all checking till then, and fail then if the type is wrong. The idea is to make static checking work until dynamic checking cuts in at runtime. May be useful for elements that can have two very different potential models (an integer or the code words "not applicable"; quantity-on-hand as a number or as the word “out-of-stock”, any Address or a more restricted “United States Address”, etc.) `$myaddress treat as element(*, USAddress)`

When an XPath Expression is Evaluated

(by an XSLT processor, for example)

The processor knows certain things to start:

- Context node (“which node am I processing now?”)
  - in XSLT, typically the node that a template matches
  - in Schematron, node named by `@context` attribute on `<rule>`
- Context size (“how many nodes am I processing with this one?”) typically the number of siblings
- Context position (“of the nodes I am processing with this one, which one is this?”)
  - Equals size of current node list (list of nodes queued up with this one)
  - First position is 1
- Other deep knowledge:
  - Values assigned to variables in scope (in XSLT)
  - All available functions
  - Namespaces in XSLT stylesheet in scope (default namespace not included)

Comparison Operators in XPath and XSLT

- XPath 1.0 defines only *general comparison* operators.
- ***General comparison operators*** compare sequences of values. In XPath 1.0, only node sequences (node sets). XPath 2.0+ have `sequence` datatype for sequences, `atomic` values for anything else.
- ***Value Comparison*** operators compare individual values, not sequences of values.
- ***Node Comparison*** operators only work of nodes and concern node equality and relationships.
- XPath 2.0+ use all 3 types (Appendix D)

Operator Meaning	**General Comparison* (for a sequence of values) *All XPath/XSLT Versions***	**Value Comparison (for single values) *XPath/XSLT 2.0 and above***	**Node Comparisons *XPath/XSLT 2.0 and above***
equal	`=`	`eq`	
not equal	`!=`	`ne`	
less than	< (as `<`)	`lt`	
less than or equal to	< =(as `<=`)	`le`	
greater than	> (as `>`)	`gt`	
greater than or equal to	>= (as `>=`)	`ge`	
equality of nodes			`is`
`left arg` follows `right arg` in document order			`>>`
`left arg` precedes `right arg` in document order			`<<`

VI. Tips, Traps, and Gotchas

(as time permits)

Why You Want to be in XPath 2.0, 3.0, 3.1

- You can write your own functions! (priceless)
- Regular Expressions!
- Lots more functions and operators
- For data: real data types, sequences, XSD-aware
- XPath 3.0: Higher order functions! (a real language)
- XPath 3.1: maps and arrays for JSON

Oxygen XPath Tools

- Are great tools!
- Let’s look at them
  - XPath window choosing version
  - Update XPath on cursor move
- XPath/XQuery Builder

Other editors and database UIs have similar tools

Quoted strings

What’s the difference between A and B?

- A. `<xsl:value-of select="title">`
- B. `<xsl:value-of select="'title'">`
- For A
    - think element node
    - think `child::`
- For B:
    - think `string`

In Attribute Values, “<” vs. “<”

(This is XML well-formedness, NOT an XPath problem!)

- “<” is an XPath operator
    - ...character not allowed in an attribute value! (XML well-formedness)
    - `<xsl:if test="@position < 10">...</xsl:if>`
      ...isn't well-formed!
- In XML attribute values, express “<” as “<”
    - `<xsl:if test="@position < 10">...</xsl:if>`
    - XML parser reports “`@position < 10`” to processor...

  ...we're fine!

Don't Neglect the Obvious

sometimes the syntax can throw you

- Spaces around operands
  - aren’t just a good idea
  - but the rule
- “ `big-dogs` ” is a name
- “ `big - dogs` ” is arithmetic on elements

Test for Content Using `normalize-space()`

- Function `normalize-space()` trims extra whitespace from a string of text
    - removes leading whitespace
    - removes trailing whitespace
    - reduces interior runs of whitespace characters to a single space
- If there’s nothing but whitespace in the string,
  then nothing (an empty string: "") remains after this trimming
- So normalize-space(self::node()) tests `true`
  only when the string tested has content besides whitespace

```
<rule context="surname">
  <assert test="normalize-space(.)">Surname has no content</assert>
</rule>
```

Very Cool: This assertion will fail for all of these:

- `<surname/>`
- `<surname>   </surname>`
- `<surname>`
  `</surname>`

(In other words, if you clean up all the whitespace and there is nothing left, the node is empty!)

Normalize Space Warning

- Use `normalize-space()`
  - for testing for empty elements
  - for any testing you want!
  - to trim space from text-only nodes
- Do NOT use `normalize-space()`
  - to trim space from mixed content nodes such as `<p>` or `<title>`
  - `normalize-space()` works on strings and all interior markup will vanish

```
<title>Why <italic>E. coli<italic> are Harmful</title>
```

becomes

```
<title>Why E. coli are Harmful</title>
```

The Axis `descendant-or-self::` is a Full Step

- You want to find the very last list item in the entire document, ignoring all other list item nodes.
- This XPath won’t do that
  `//list-item[last()]`

Why not? Let’s look at what that XPath means:

- The long form of that XPath is:
  `/descendant-or-self::node()/child::list-item[position()=last()]`
- What this means: There are two steps, and the predicate only filters the second step
  - first all the descendant nodes are found
  - then, for each one, the last child `list-item` is found

How do we solve it?
`(//list-item)[last()]`
(group the nodes with parentheses and apply the predicate to the whole group)

Union Operator (“`|`”) vs. Boolean Operator (“`or`”)

Do these do the same thing? (Why or why not?)

- `<xsl:if test="title | body">...</xsl:if>`
  This is a union operator
- `<xsl:if test="title or body">...</xsl:if>`
  This is a boolean

How about these?

- `<xsl:if test="title='Preface' or body">...</xsl:if>`
  An `xsl:if` test on a string with any content is always true
- `<xsl:if test="title='Preface' | body">...</xsl:if>`
  (Union of a string and a nodeset is always an error)

Say It Ain’t So!

`!=` operator can lead to non-intuitive results: `not()` is usually safer.

- `select="slide[@type!='intro']"`
  - Selects `slide` children (of the current node) that have a `@type` attribute, where the value is NOT “intro”.
  - Gotta have that attribute!
  - If `@type` returns empty node set, it tests `true` as not equal to “intro”
- `slide[not(@type='intro')]`
  - Selects `slide` children (of the current node) that do not have a `@type` attribute whose value is “intro”.

Sequences Need their Boundaries! (optional)

Commas and parentheses make sequences,
and sequences are *a single* thing

For example:

- The function `min()` returns the minimum value *in a number or sequence*
- `min((8, 5, 23))` returns `5`
- `min(6)` returns `6` (six is just a number)
- But `min(8, 5, 23)` would return an error
  `min()` needs a sequence, and we're giving it three numbers and some commas

Mulberry Quick Refs

Take one of each and take a look!

Advanced Tips and Gotchas (optional)

Select All Nodes Except

- In XPath 1.0: `*[not(self::title)]`
- In XPath 2.0 and 3.0:
  `(* except title)`

How to select an empty node set:

- `/..` or
- `@text()`

Some XPath 2.0 and 3.0+ Expressions

that *behave* like document-order, no-duplicate node sets

- Expressions that use the path operator “ `/` ”
- Expressions that reference an axis
- Expressions using the operators:
  - union (`|`)
  - `intersect`
  - `except`

Be Careful for *Context*

`For` Expressions and Location Paths are Different!

Location Paths	**`For ... in ... return` expressions**
Work with nodes	Work on any sequence
Duplicates eliminated	Duplicates allowed
Sorts results into document order	No sort, input order retained
Each step is evaluated in turn, resetting context node	Does not set context node

```
sum(for $n in child::name return concat($n/fname, ' ', $n/surname))
```

- Warning: the context for the ***`return`*** is the same as the context for the whole ***for***

- So this will *not* work as intended:

  ```
  for $n in child::name return concat(fname, ' ', surname)
  ```

- Fix this with

```
for $n in child::name return concat($n/fname, ' ', $n/surname)
```

Surprise! Operators Can Force Document Order

- You've sorted some employee records into a sequence “`$sorted-employees`”
- Now that you have them, you want just the names
- The location path:

  `$sorted-employees/name`

  - would return the names *in document order* not sorted
  - because it contains a “`/`”
  - (with thanks to Michael Kay for this example)
- You probably want

```
for $e in $sorted-employees return $e/name
```

How to Use Types in a Type-free World

(DTD-valid or well-formed, for example)

You do not want something dealt with as “untyped-atomic”,
but you don’t have a schema.

Either:

- Cast a few types
  - `cast` starts with an existing value and creates a new value of the specific type
  - Syntax
    source-type **cast as** *target-type*
- Or make types using constructor functions
  `xs:date("2005-08-30")`

VII. Colophon

- Slides and handouts created from single XML source
- Slides projected from HTML generated from XML using XSLT
- Print copy created from the same XML source
  - XSLT transform generates XHTML
  - Antenna House Formatter makes PDF from:
    - XHTML
    - CSS3 (slightly extended)
    - Graphics sizing table

Appendix A

Answers to Short/Full Syntax Exercise

Full Syntax	Abbreviated Syntax
`self::node()/child::PROLOGUE/child::TITLE`	`./PROLOGUE/TITLE`
`/descendant-or-self::node()/child::STAGEDIR`	`//STAGEDIR`
`child::*/child::LINE`	`*/LINE`
`parent::node()/child::processing-instruction("foo")`	`../processing-instruction("foo")`
`attribute::bar`	`@bar`

Abbreviated Syntax	Full Syntax
`PERSONA`	`child::PERSONA`
`./PGROUP`	`self::node()/child::PGROUP`
`//FM/P`	`/descendant-or-self::node()/child::FM/child::P`
`/`	`/`
`SCENE/LINE`	`child::SCENE/child::LINE`
`../TITLE`	`parent::node()/child::TITLE`

Appendix B

Pattern Matching in XSLT and Schematron

Subset of XPath Used for Matching/Testing

A subset of XPath expressions are used in XSLT, Schematron, and elsewhere for matching. This is an application of XPath that is defined in the XSLT/ Schematron specifications. When location paths are used as patterns, the processor has already selected a node and the question is whether the node matches the pattern. Basically matching works as follows:

- You have a node (an XSLT or Schematron engine or similar got it for you)
- You have an XPath expression called a "pattern"
  - possibly as an XSLT `<xsl: template match="`*`pattern`*`"`
  - possibly as a Schematron `<rule context="`*`pattern`*`"`
- The question is: "does the node you have match that pattern?"
- The answer is a boolean, `true` or `false`

Some Pattern Matching Examples

`<xsl:template match="para"/>`

Matches every element named `para`

`<xsl:template match="*"/>`

Matches any element

`<xsl:template match="SECTION/TITLE"/>`

Matches any element named `TITLE`, but only when the title is a child of `SECTION` element

`xsl:template match="employee[@category='critical']"/>`

Matches any element named `employee` that has an attribute named "`category`" that has a value of "`critical`"

Two Ways to Read the Same Location Path

The same XPath syntax can have a different meaning and reading depending on where it is used. When an XPath location path is used as a “match pattern” it is read and evaluated very differently from the *same* location path used as an expression, for example as the value of a `select` attribute. As an example, take the XPath expression

```
slide/title
```

As a match pattern, it matches any `title` element that is the child of a `slide`. Patterns work right to left, testing one node at a time. (Are you a title? Is your parent a slide?) The expression returns a Boolean: true or false.

As a location path, the expression is evaluated in relationship to the context node (it is the short syntax form of `child::slide/child::title`). It returns not a Boolean but a node set, “the `title` children of the `slide` children of the context node”. Location paths are evaluated left to right, so, when evaluated relative to the `segment` context node, this path selects the `title` children of the `slide` children of `segment`. It goes like this:

- Find the `segment` (the context node)
- Get the `slide` children of that `segment`,
- Then get the `title` children of those slides
- Return a set of nodes (e.g., the selected `title`s)

Cheat Sheet: Location Paths in `select` Attributes

The table below provides samples of location path syntax when applied in an XPath select expression. The table after this one illustrates many of these same expressions as they are used in an XSLT “match” pattern.

Each expression is evaluated relative to an already-selected context node and returns a node set.

Expression	Returns
`name`	`name` children of the context node
`/`	Root node
`.`	The context node itself (equivalent to `self::node()`)
`..`	The parent of context node (equivalent to `parent::node()`)

Expression	Returns
`./name`	`name` children of the context node (equivalent to `name` and to `child::name`)
`.//name`	`name` descendants of the context node
`//name`	`name` descendants of the root node
`name1 | name2`	Union of `name1` and `name2` children of context node
`../name`	`name` children of parent of context node (i.e., `name` sibling elements, and context node if context node is `name`)
`../@name`	`name` attribute of parent of context node
`*`	All element children of context node
`@*`	All attributes of context node
`*/name`	All `name` grandchildren (i.e., `name` children of element children) of the context node
`name1/name2`	All `name2` children of `name1` children of the context node
`name1//name2`	All `name2` descendants of `name1` children of the context node. Includes all `name2` children of `name1` children of the context node
`//name[1]`	All `name` descendants of the root, that are the first `name` child of their parents. Different from `/descendant::name[1]` (the first `name` descendant of the root)

Location Paths in `match` Attributes

A match pattern specifies a set of conditions on a node. “A node matches a pattern if the node is a member of the result of evaluating the pattern as an expression with respect to some possible context”. The idea is that some process (the XSLT processor) has already selected a node. Matches act as tests on that node.

These expressions return a boolean `true` or `false`, either the node you have matches the pattern or it does not.

Pattern	Matches
`name`	Any `name` element
`/`	The root node
`*`	Any element node
`@*`	Any attribute node
`name1/name2`	Any `name2` element with `name1` parent element
`name1//name2`	Any `name2` element with `name1` ancestor element
`name1 | name2`	Any `name1` element or `name2` element
`text()`	Any text node
`node()`	Any node that is a child of another node (i.e., because of implicit `child::` axis specifier, not the root or an attribute node)
`id("xx")`	The element with the unique ID “xx”
`name[1]`	Any `name` element that is the first `name` child of its parent
`@name`	Any `name` attribute
`*[position()=1]`	Any element that is the first child of its parent

Match Patterns are a Subset of XPath Expressions

Patterns have been designed as a subset of XPath expressions (more particularly, of XPath expressions *that return node sets*), and they have a few restrictions that do not apply to location paths in general.

Patterns may only look “down” the tree, so they may use `/`, `//`, `child::`, or `attribute::` axes. By the same reasoning therefore, they may *not* contain:

- Axis names other than `child::` and `attribute::` (e.g., `preceding-sibling::` not allowed)
- `.` (`self::node()`)
- `..` (`parent::node()`)
- Variable or parameter references

But a pattern *may* include

- `|` union operator (e.g., `match="name | url"`)
- `/` operator (e.g., `match="slide/title"`)
- `//` operator (e.g., `match="//title"`)
- Predicates (as long as they contain no variable references)

Patterns may also use the `id()` or `key()` functions (though again, without variable references).

Appendix C

A Few XPath Functions

Number Functions

XPath deals with numbers (Like 1, 2 and 8) and converts things like strings into numbers. XPath numbering includes:

- Positive and negative numbers
- Not-a-Number (NaN)
- Positive zero
- Negative zero
- Positive infinity
- Negative infinity

The function `number(`*`expr`*`)`, when asked to convert:

1. Number: produces the number
2. String: if parses as number, convert, otherwise NaN
   "Debbie" versus "42"
3. Boolean: true=1, false=0
4. Node-set: convert to string, then evaluate

`number(`*`expr`*`)` Examples

Expression (convert to a number)	Returns	Rule
`number(42)`	42	#1 Number
`number(1 > 2)`	0	#3 Boolean
`number("XPath")`	NaN	#2 String
`number("42")`	42	#2 String

Numeric operations include:

- Addition, subtraction, division, rounding, etc.
  - `5 + 2` returns `7`

  - `round(13 div 3)` returns `4`
- **Warning**: division operator is `div`, not `/`
- Use `mod` for a remainder, e.g.,
  - `5 mod 2` returns `1`
  - `6 mod 2` returns `0`

Numeric Expressions

`+`	Add arguments
`-`	Subtract arguments
`*`	Multiply arguments
`div`	IEEE 754 floating point division
`mod`	Return remainder from integer division operation
`ceiling(`*`expr`*`)`	Return smallest (closest to negative infinity) integer not less than *`expr`*
`floor(`*`expr`*`)`	Return largest (closest to positive infinity) integer not greater than *`expr`*
`round(`*`expr`*`)`	Return integer closest to *`expr`*. If two such numbers, return number closer to positive infinity.
`sum()`	Sum values of nodes in node-set

Numeric Function Examples

Expression	Returns
`1 + 1`	2
`1 - 1`	0
`2 * 2`	4
`9 div 2`	4.5
`9 mod 2`	1
`floor(4.5)`	4
`ceiling(4.5)`	5

Expression	Returns
`round(4.5)`	5
`floor(-4.5)`	-5
`ceiling(-4.5)`	-4
`round(-5.5)`	-5
`round(5.5)`	6

String Functions (type xs:string)

String functions are probably the most commonly used in XPath for documents. You can compare strings, concatenate strings, make upper case into lower (or reverse), and such like. Strings are just sequences of characters (UCS [Universal Character Set] characters, using the same character set that the XML Recommendation uses.)

In XML, pretty much everything is a string, but you can use the `string()` function to convert other objects to strings. XPath 1.0 will coerce things into strings if a string function is used.

Warning for programmers: Substring expressions count first character as 1 (one), not 0 (zero)!

When an object is converted into a string:

- Sequence of nodes: return value of first node, or empty string if empty node-set
- Number: return string in form of number (“42”)
  - NaN returns "`NaN`"
  - Positive zero returns "`0`"
  - Negative zero returns "`0`"
  - Positive infinity returns "`infinity`"
  - Negative infinity returns "`infinity`"
- Boolean: return "`false`" if false, return "`true`" if true

All of the XPath 1.0 String Functions

`concat(`*`$string`*`...)`	Return concatenation of arguments

`contains($string1, $string2)`	Return true if first argument string contains second argument string, otherwise false
`normalize-space($string)`	Return argument string after stripping leading and trailing white space and reducing multiple white-space characters to single space. Works only on strings!
`starts-with($string1, $string2)`	Return true if first argument string starts with second argument string, otherwise false
`string-length($string?)`	Return number of characters in the string. Argument defaults to string value of context node.
`substring($string, $number, $number?)`	Return substring of first argument starting at second argument with length specified by third argument
`substring-after($string1, $string2)`	Return substring of first argument string following first occurrence of second argument string in first argument string, otherwise return empty string
`substring-before($string1, $string2)`	Return substring of first argument string preceding first occurrence of second argument string in first argument string, otherwise return empty string
`translate($string1, $string2, $string3)`	Return first argument string with occurrences of second argument string replaced by corresponding characters from third argument string

String Examples for the Functions Just Described

Expression	Returns
`concat("Four ", "score ", "and seven")`	`"Four score and seven"`
`contains("Four score and seven", "core")`	True
`contains("Four score and seven", "four")`	False
`normalize-space(" foo
 bar ")`	`"foo bar"`
`starts-with("foo", "f")`	True

Expression	Returns
`starts-with("bar", "f")`	False
`string-length("Four score and seven")`	20
`substring("Four score and seven", 4, 7)`	`"r score"`
`substring-after("Four score and seven", "core")`	`" and seven"`
`substring-after("Four score and seven", "four")`	`""`
`substring-before("Four score and seven", "core")`	`"Four s"`
`substring-before("Four score and seven", "four")`	`""`
`translate("bar", "abc", "ABC")`	`"BAr"`
`translate("EN-us", "ABCDEF...YZ", "abcdef...yz")`	`"en-us"`
`upper-case("iso sts")`	`ISO STS`
`matches("Ides of March", "Ides | April")`	`true`
`tokenize('March 15, 44BCE','([ ] | ,)+')`	`('March', '15', '44BCE')`
`replace('March 15, 44BCE','BC[E]?' , ' before the Common Era')`	`'March 15, 44 before the Common Era'`

Selected XPath 2.0 and 3.0 String Functions

`upper-case($string)`	Translates each character to uppercase (or returns it unchanged if there is no equivalent)
`lower-case($string)`	Translates each character to lowercase (or returns it unchanged if there is no equivalent)

`compare($string1,$string2,$collation?)`	Returns which string (of two strings given) appears first in a given collation (or the processor's default collation)
`ends-with($string1,$string2)`	Like `starts-with()` (still in XPath 2.0) except inspecting the end of a string
`string-join($sequence,$separator)`	Concatenates all the strings given in a sequence, using an optional separator between adjacent strings
String Expressions Using Regular Expressions	
`matches($string,$regex,$flags?)`	Returns boolean to indicate if string matches regular expression; matches if any substring matches (unless an anchor ^ or $ is used)
`replace($string,$regex,$replacement,$flags?)`	Constructs an output string by replacing parts of the input string that match regex (while copying non-matching substrings); replacement string can reference matched substrings
`tokenize($string,$regex,$flags?)`	Splits a string into a sequence of substrings (tokens) as delimited by separators that match the regex

String Examples for the Functions Just Described

Expression	Returns
`upper-case("iso sts")`	`ISO STS`
`lower-case("ISO STS")`	`iso sts`
`compare('abc', 'abc)'`	`0`
`compare('abc', 'def')`	`-1`
`ends-with("Mulberry", "berry")`	`true`

Expression	Returns
`string-join( ('John', 'Paul', 'George', 'Ringo'), "!")`	`John!Paul!George!Ringo`
`matches("Ides of March", "Ides | April")`	`true`
`replace('March 15, 44BCE','BC[E]?' , ' before the Common Era')`	`'March 15, 44 before the Common Era'`
`tokenize('March 15, 44BCE','([ ] | ,)+')`	`('March', '15', '44BCE')`

Boolean Functions

- Boolean objects can have two values
    - `true`
    - `false`
- Operators include
    - `and`
    - `or`
    - comparison operators (e.g., <, >=)
    - equality operators (=, `!=`)
- Function `boolean(`*`expr`*`)`converts the required argument to a boolean:
    - Number: true iff not positive zero, negative zero or NaN (Not a Number)
    - Node-list: true iff non-empty
    - String: true iff length is non-zero

Boolean Function Examples

Expression	Returns
`boolean(1)`	True
`boolean(1 + "XSL")`	False
`boolean("XSL")`	True

Expression	Returns
`boolean("")`	False

Boolean Functions

`not(`*`expr`*`)`	Returns true if argument false, and false otherwise
`true()`	Returns true
`false()`	Returns false
`lang(`*`string`*`)`	Returns true if *`string`* matches language of current (Case insensitive!)

Sequence of Nodes (Node Set) Functions

- Location paths can be used as expressions
- Result is node set selected by path
- “*`node-set`*`|`*`node-set`* ” returns union of node-sets
- “*`node-set`*`[`*`expr`*`]`” filters node-set

Node Set Functions

`count(`*`node-set`*`)`	Returns number of nodes in *`node-set`*
`id(`*`object`*`)`	Returns node-set containing element in same document with ID equal to any token in string value of *`object`*
`last()`	Returns number equal to context size
`local-name(`*`node-set?`*`)`	Returns local part of name of first node in *`node-set`*
`name(`*`node-set?`*`)`	Returns combined prefix, colon, and local part of first node in *`node-set`*
`namespace-uri(`*`node-set?`*`)`	Returns namespace of name of first node in *`node-set`*
`position()`	Returns number equal to context position. First position is 1, last equal to `last()`

XPath 2.0 and 3.0 Functions for Sequences

There are Bunches of Functions for Sequences

- Basic list manipulation
  - `insert-before($sequence,$position,$insertion)` and `remove()`
  - `reverse($sequence)`
  - `index-of($sequence,$item,$collation?)` returns position of `$item` in `$sequence` (starting at 1)
    - a collation may be used to affect string comparison
  - `distinct-values($sequence)` returns the distinct values in the sequence (de-duplicates values)
  - `subsequence($sequence,$start,$length?)` — like `substring($sequence,$start,$length?)` for sequences
- Test cardinality in sequences
- `deep-equal($sequence1,$sequence2)` (are these sequences pair-wise really, really equal)
- Perform math on items in a sequence
  - `count($sequence)`
  - `average($sequence)`
  - `max($sequence)`
  - `min($sequence)`
  - `sum($sequence)`

As well as sequence generation functions dealing with IDs and IDREFs, document availability testing, and document collections

Numerous Functions for Durations, Date and Time

- Addition and subtraction of dates and durations
- Multiplication and division on a few types
- Timezone adjustments

- Comparisons (less-than, greater-than, equal) for: `date`, `month`, `time`, `timeDuration`, `YearMonth`, `MonthDay`, etc.

XPath 2.0 and 3.0+ Quantified Expressions

Quantified expressions use the operators `some` and `every`.

- They indicate whether an expression satisfies these conditions?
- Both return a boolean; it satisfies or it does not
- `some`: test if at least one item in expression satisfies the condition

  ```
  some $variable  in expression  satisfies expression
  ```

- `every`: tests if all values in expression satisfy the condition

  ```
  every $variable  in expression  satisfies expression
  ```

As an example:

```
some $x in /students/student/name satisfies $x = "Steve"
```

(With thanks to Evan Lenz for the example)

XPath 2.0 and 3.0+ have Conditional Expressions

`if ... then ... else...`

- Evaluate an expression
- If true, evaluate `then` branch
- If false, evaluate `else` branch
- Then return the result of the evaluation
- Syntax

  ```
  if (test-expression)
  then expression
  else expression
  ```

- Example

  ```
  if ($part/@discounted)
  then $part/wholesale
  else $part/retail
  ```

XPath Operations

Comparison Operators in XPath and XSLT

- XPath 1.0 defines only General Comparison operators.
- General Comparison operators compare sequences of values. (XPath 1.0 has only node sequences/nodesets; XPath 2.0 and 3.0 have a sequence datatype for sequences of nodes, atomic values, anything.)
- Value Comparison operators compare individual values (not a sequence of values, only a single-item sequence)
- Node Comparison operators work only on nodes and concern node equality and relationship between the nodes in the tree.
- XPath 2.0 and 3.0 have all three comparison types.

Operator Meaning	**General Comparison* (for sequences of values)** **All XPath versions**	**Value Comparison (for single values)** **XPath 2.0 and 3.0**	**Node Comparisons** **XPath 2.0 and 3.0**
equal	=	eq	
not equal	!=	ne	
less than	< (as <)	lt	
less than or equal to	<= (as <=)	le	
greater than	>	gt	
greater than or equal to	>=	ge	
equality in nodes			is
left arg follows right arg in document order			>>

Operator Meaning	General Comparison* (for sequences of values) All XPath versions	Value Comparison (for single values) XPath 2.0 and 3.0	Node Comparisons XPath 2.0 and 3.0
left arg before right arg in document order			<<

* If you have old XSLT 1.0 programs, they may run unchanged in XSLT 2.0 and 3.0. If there are type errors, in XSLT 2.0 and above, the “XSLT 1.0 compatibility switch” can make General Comparisons work almost exactly as they do in XSLT 1.0. Without the compatibility switch, there are some differences in when and how values of one type are converted to values of another type for comparison.

The next few pages explain all these operators in more detail.

Several Types of Operators Over Items

- Arithmetic operators
- Boolean operators
- Node comparison operators
- Comparison operators, which may be considered as two types:
  - Value comparisons
  - General comparisons

Arithmetic Operators

Arithmetic operators are just what you’d expect from elementary math class. They handle the simple operations like addition and subtraction. Arithmetic operators are used on:

- numbers (`xs:integer`, `xs:decimal`, etc.)
- on dates and durations too.

Operator	**Operation**
+	Addition
-	Subtraction

`*`	Multiplication
`div`	Division
`idiv`	Integer Division
`mod`	Modulo

Boolean Operators

There are two boolean operators: “ `and` ” and “ `or` ”, which compare expressions and return boolean values of “ `true` ” or “ `false` ”. Conterintuitively, there is no “ `not` ” operator; the not function (forgive the pun) is provided as a function, `not()` rather than as an operator.

- A series of booleans can be strung together:
  `(x or y or z or w or j or d or q)`
- Parenthesis may be used as needed.
- The `and` operator is of higher priority than the `or` operator, so `(x and y or a and b)` would resolve to
  `( (x and y) or (a and b) )`

Operator	**Operation**
`and`	Returns “ `true` ” if the two expressions it connects are both true
`or`	Returns “ `true` ” if either of the two expressions it connects is true
`not()`	Not an operator. The `not()` function returns “ `true` ” if the argument is false

Node Comparison Operators

Since nodes now come in ordered list instead of sets, it is possible to compare any two nodes, and there are node comparison operators to make that possible. These operators can be used to compare two nodes:

- by identity, or
- by document order

The general syntax is as follows, with the operator used between two node operands:

```
leftoperand  operator  rightoperand
```

Operator	Operation
`is`	True if operands have the same identity, otherwise false
`<<`	True if the left operand precedes the right (in document order), otherwise false
`>>`	True if the left operand follows the right (in document order), otherwise false

Operators for Combining Sets of Nodes

- Uses sequences to simulate node sets
  - Results are returned in document order
- Given two sequences of nodes:

union (“`|`”)	Include a node in the result if it is present in either sequence
`intersect`	Include a node in the result if it is present in both sequences (all items in common)
`except`	Include a node in the result if it is present in the first sequence but not the second (difference between)

`except` — The `except` operator can make code much easier to read. For example the convoluted XPath 1.0 expression:

```
child::*[not(self::p)]
```

Can be done easily in XPath 2.0 and 3.0+ as:

```
(child::* except child::p)
```

`intersect` — returns `pb` elements preceding the context inside the same (closest) `div`:

```
(preceding::pb intersect ancestor::div[1]//pb)
```

- Given the sequence `$nodes` = `(para, list, table, figure)`
  - Short for `(child::para, child::list, child::table, child::figure)`
  - All `para` children, then all `list` children, then all `table` children, then all figures...

  - ... in that order
- All these will sort this sequence back into document order!
  - `$nodes | $nodes`
  - `$nodes | ()`
  - `$nodes intersect $nodes`
  - `$nodes except ()`
  - `$nodes/.`

Value Comparison Operators

These values are used for atomic values, replacing the XPath 1.0 operators (=, !=, <, >, >=) which are used for sequences. They may be more useful when dealing with untyped data. Value comparison operators are:

- Used to compare single values
- May be used on numbers (`xs:integer`, `xs:decimal`, etc.).
- Result in `true` or `false`

Operands are “atomized” before comparison

- An empty sequence returns an empty sequence
- More than one value is an error

```
//product[weight gt 100]
```

Operands are “atomized” before comparison

Table of Value Comparison Operators

Operator	**Operation**
`eq`	Equal
`ne`	Not equal
`lt`	Less than
`le`	Less than or equal
`gt`	Greater than
`ge`	Greater than or equal

Atomization

The process of atomization is used to turn a sequence into a sequence of atomic values. This may occur in arithmetic expressions, comparison expressions, function calls, or casting expressions. The process is applied to each item in a sequence, with the result being either

- a sequence of atomic values, or
- a type error.

The process works essentially like this. Each item in a sequence is examined and

- if it is an atomic value, uses that value,
- if it is a node, uses its typed value, or
- if it is neither, returns an error.

General Comparison Operators

The general operators are the ones that used to be used in XPath 1.0 (`=, !=, <, >, >=`). In XPath 2.0, one important distinction is that either side of the expression between the operators can be *an expression* instead of just a value. The general comparison operators:

- May compare values or sequences
- Result is `true` or `false.`
- Before comparison, atomization is applied to each operand, producing a sequence of atomic values.
- Rules are different under backwards compatibility mode.

Another major difference is that these operators working *on untyped data* work differently in XPath 2.0 than they did in XPath 1.0. In XPath 1.0, nodes did not have types. What happened in a node comparison depended on what kind of operator was being used and whether the node value was convertible to, for example, a number. (The string "42" can convert to an integer, the string "Debbie" cannot.) In XPath 1.0, if you asked if “ `a < b` ” and a was “ `3` ” and b was “ `10` ”, the comparison would be done as if the a and b were both numeric, and the answer would be `true`. In XPath 2.0, if a and b are untyped, they will be treated as strings. So “ `a < b` ” is are not compared numerically, and it is “ `false` ”.

Table of General Comparison Operators

Operator	**Operation**
`=`	Equal
`!=`	Subtraction
`<`	Multiplication
`<=`	Division
`>`	Integer Division
`>=`	Modulo

Built-in Operator Precedence: Beyond My Dear Aunt Sally

XPath 2.0 Operators have *built-in precedence*

- If precedence is equal proceed left-to-right
  - `(x + y - z)` is really
  - `(x + y) - z`
- Higher items (in the chart on the next slide) bind before lower items
  - `x or y and z` is really
  - `x or (y and z)`
- Items of a lower precedence cannot be contained by operators of a higher precedence

Operator Precedence

Operators listed from *highest to lowest*
(commas act as separators between operators below)

- `( )`, `[ ]`, `{ }`
- `/`, `//`
- `?`, `*`(as an occurrence indicator), +(same)
- `-`(unary), +(unary)
- `cast`

- `castable`
- treat
- `instance of`
- `intersect`, `except`
- `union`, `|`
- `*`, `div`, `idiv`, `mod`
- `+` , `-`
- `to`
- `eq`, `ne`, `lt`, `le`, `qt`, `=`, `!=`, `<`. `<=`, `>`, `>+`, `is`, `>>`, `<<`
- `and`
- `or`
- `for`, `some`, `every`, `if`
- `,` (comma)

Appendix E

XPath 2.0 and 3.0 Data Model (advanced, optional)

Data Model for XPath 2.0, XPath 3.0, and XPath 3.1 has three conceptual building blocks

- Trees made up of nodes (just like XPath 1.0)
- Atomic values (integers, strings, booleans, etc.)
- Sequences of “items”
  - an item is an atomic value or a reference to a node
  - each item has a value and a type (`xs:integer`, `xs:string`, etc.)
  - a single item is considered to be a sequence containing one item
  - a sequence cannot be a member of a sequence

(Why define atomics and sequences? Because atomics and sequences represent intermediate results during expression processing!)

Sequences

- Location paths in XPath 1.0 return node sets
- Location Paths in XPath 2.0 return sequences
- Node sets
  - have no duplicates
  - have no intrinsic order
- Sequences
  - are an ordered collection (list)
  - of zero, one, or more *items* (not just nodes)
  - may well have duplicates

In XPath 1.0 there were
“sets” of “nodes”

- XPath 1.0 centered its view on an XML document as a tree of nodes

- Nodes have identity
- Node sets are *unordered* collections of nodes
  - usually fall back to document order
  - sometimes (reverse axes) use reverse document order
- Nodes (and their subtrees) can be copied, but references to them cannot be multiples

In XPath 2.0/3.0+ there are
“sequences” of “items”

- XPath 2.0 does not center on a single document tree, but on arbitrary data sets
- These can be arranged in “sequences” of “items”
  - sequences are lists, ordered sets of
    - pointers to nodes (which still have identity) and
    - simple-typed *values*
  - may contain duplicates
- `count($node-set) = count($node-set | $node-set)` is still true (due to semantics of “`|`”, the union operator)
- But now we can also say `($node-set , $node-set)`
  - A sequence of all the nodes in `$node-set`, *then all the same nodes again*

XPath 2.0 and 3.0 are All about sequences. A sequence is an ordered collection of zero or more items:

- All expressions return sequences
- All values are in sequences
- A singleton is a one-item sequence
- The empty sequence is a valid sequence
- Members of sequences (unlike nodes) do not have identity
- All sequences are ordered

- Duplication is allowed inside sequences!
- Sequences cannot nest (one level only)
- if `$seq = (x, y, z)`,
- then `(a, b, $seq, y, c)` evaluates to
- `(a, b, x, y, z, y, c)`

Examples of Sequences

- A document root (and therefore a document)
- One node (and therefore a subtree)
- A series of nodes and/or document roots
- A string value (like “42”)
- An integer value (like 42)
- A series of strings, integers, and/or nodes
- A set of nodes described by an XPath expression, in an order
- The results of evaluating an XPath expression (say, a series of strings or dateTime values)

(All the world’s a sequence!)

Constructing Sequences

- The comma operator “`,`”
    - means concatenation (of items, not strings)
    - makes sequences: `(a, 1, w)`
- Members of sequences (unlike nodes) do not have identity
- Sequences cannot nest (one level only)
    - if `$seq = (x, y, z)`, then `(a, b, $seq, y, c)` evaluates to
    - `(a, b, x, y, z, y, c)`
- Remember, duplication is allowed inside sequences!

Sequences can Contain Atomic Types

- Identified by the namespace: `xmlns:xs="http://www.w3.org/2001/XMLSchema"`
- A type derived from another atomic type in a schema, *by restriction*
- All XSLT processors support a minimal set, *even without a Schema*:
  - `xs:boolean`
  - `xs:decimal`
  - `xs:double`
  - `xs:integer`
  - `xs:string`
  - `xs:QName`
  - `xs:anyURI`
  - `xs:dayTimeDuration`
  - `xs:date`
  - `xs:time`
  - `xs:dateTime`
  - `xs:yearMonthDuration`
  - `xs:anyAtomicType`
  - `xs:untyped`
  - `xs:untypedAtomic`

(May also support other W3C XML Schema primitive types)

Expressions for Sequences

Constructing Sequences

The comma operator “`,` us used to create sequences, for example, `(a, 1, w)`”

- means concatenation (of items, not strings)

- The sequence `(p, list, table, figure)`
  - these are nodes in a tree
  - they still have axes
  - all p children followed by all list children, followed by all tabled children, followed by all figure children

Another way to construct sequences uses the “**`to`** operator:”

`expression` **`to`** *`expression`*

- Each expression must evaluate to an integer
- first integer must be smaller than the second
- Makes consecutive integers in ascending order

`(1 to 10)`	makes `(1,2,3,4,5,6,7,8,9,10)`
`(10, 1 to 3)`	makes `(10, 1, 2, 3)`
`1 to count($some-sequence)`	Returns the position number of each item in the sequence `$some-sequence`
`reverse(5 to 10)`	Evaluates to `(10, 9, 8, 7, 6, 5)`

Sequences Take Filters

Like predicates on paths, sequences can be filtered using “`[ ]`”

- Predicates come in two styles
  - numeric: e.g. `$seq[3]`
    - predicated value is a number; returns item in that position
    - i.e., indexes into the sequence
  - boolean: e.g. `$seq[@rating = 'good']`
    - keep any item, for which predicate tests true
    - `$seq[position()=3]` is numeric predicate as boolean
- The original order is retained
- `(p, list, table)[descendant::note]`
  A sequence of all the ps, lists, and tables. but only if they have note descendants.

Iterate Over Sequences Using `for` Expressions

for $*variable* **in** *sequence* **return** *expression*

- Performs iteration over sequences
- Like XSLT `<xsl:for-each>` except inside an XPath expression
- Apply an expression to every item in a sequence
- Returns a sequence of the items returned by the mapped expression
- Can work across multiple sequences
- Both 1-to-1 mapping and 1-to-many mapping are possible

```
for $n in child::name
return concat($n/fname, ' ', $n/surname)

for $id in distinct-values(//@idref)
return count(key('elements-by-id',$id))

for $d in (0 to 6)
return (current-date() +
($d * xs:dayTimeDuration('P1D')))

sum(for $i in order-item return $i/@price * $i/@qty)
```

Sorting into document order

- Given the sequence `$nodes` = `(para, list, table, figure)`
  - Short for `(child::para, child::list, child::table, child::figure)`
  - All `para` children, then all `list` children, then all `table` children, then all figures...
  - ... in that order
- All these will sort this sequence back into document order!
  - `$nodes | $nodes`
  - `$nodes | ()`
  - `$nodes intersect $nodes`
  - `$nodes except ()`
  - `$nodes/.`

`ancestor` Axis Example

Axis specifier	Node set
`ancestor::`	`article-meta`, `front`, `article`, `/`

`ancestor-or-self` Axis Example

Axis specifier	Node set
`ancestor-or-self::`	`article-categories`, `article-meta`, `front`, `article`, `/`

Appendix H

`child` Axis Example

Axis specifier	Node set
`child::node()`	`subj-group`

`descendant` Axis Example

Axis specifier	Node set
`descendant::`	`subj-group`, `subject`, `"Methodology"`

`descendant-or-self` Axis Example

starting context:
`/descendant::article-categories`

○ context node

✗ node(s) selected

evaluating
`descendant-or-self::node()`

Axis specifier	Node set
`descendant-or-self::`	`article-categories`, `subj-group`, `subject`, `"Methodology"`

`following` Axis Example

Axis specifier	Node set
`following::`	`title-group`, `article-title`, `"Music Notation` ...", `contrib-group` ... and all its descendants ... , `aff` (with `@id="l1"`), `label`, `1`, `"Department of` ...", `aff` (with `@id="l2"`) ... and all its descendants ... , `<?more?>` processing instruction, `<?more?>` processing instruction, `<?more?>` processing instruction

Appendix L

`following-sibling` Axis Example

Axis specifier	Node set
`following-sibling::`	`title-group`, `contrib-group`, `aff` (with `@id="l1"`), `aff` (with `@id="l2"`), <?more?> processing instruction

Appendix M

`parent` Axis Example

Axis specifier	Node set
`parent::`	`article-meta`

Appendix N

`preceding` Axis Example

Axis specifier	Node set
`preceding::`	`"10.1186/1742-9994-3-18"`, `article-id` (with `@pub-id-type="doi"`), `"17112384"`, `article-id` (with `@pub-id-type="pmid"`), `1742-9994-3-18`, `article-id` (with `@pub-id-type="publisher-id"`), `"London`, `publisher-loc`, `"BioMed Central"`, `publisher-name`, `publisher`, `"1742-9994"`, `issn`, `"Frontiers in Zoology"`, `journal-title`, `"Front Zool"`, `journal-id`, `journal-meta`

Appendix O

`preceding-sibling` Axis Example

starting context:
`/descendant::article-categories`

○ context node

✕ node(s) selected

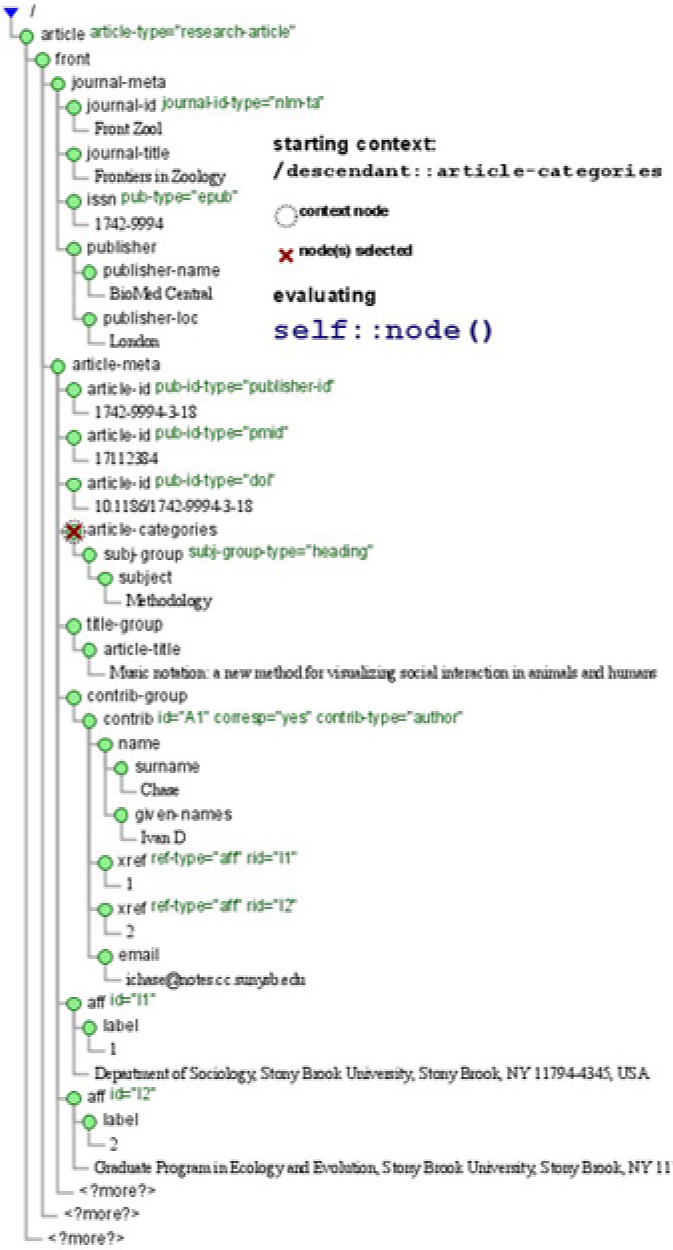
evaluating
`preceding-sibling::node()`

```
/
  article article-type="research-article"
    front
      journal-meta
        journal-id journal-id-type="nlm-ta"
          Front Zool
        journal-title
          Frontiers in Zoology
        issn pub-type="epub"
          1742-9994
        publisher
          publisher-name
            BioMed Central
          publisher-loc
            London
      article-meta
      ✕ article-id pub-id-type="publisher-id"
          1742-9994-3-18
      ✕ article-id pub-id-type="pmid"
          17112384
      ✕ article-id pub-id-type="doi"
          10.1186/1742-9994-3-18
      ○ article-categories
          subj-group subj-group-type="heading"
            subject
              Methodology
        title-group
          article-title
            Music notation: a new method for visualizing social interaction in animals and humans
        contrib-group
          contrib id="A1" corresp="yes" contrib-type="author"
            name
              surname
                Chase
              given-names
                Ivan D
            xref ref-type="aff" rid="I1"
              1
            xref ref-type="aff" rid="I2"
              2
            email
              ichase@notes.cc.sunysb.edu
        aff id="I1"
          label
            1
          Department of Sociology, Stony Brook University, Stony Brook, NY 11794-4345, USA
        aff id="I2"
          label
            2
          Graduate Program in Ecology and Evolution, Stony Brook University, Stony Brook, NY 11794-5245, USA
        <?more?>
      <?more?>
    <?more?>
```

Axis specifier	Node set
`preceding-sibling::`	`article-id` (with `@pub-id-type='doi'`), `article-id` (with `@pub-id-type='pmid'`), `article-id` (with `@pub-id-type='publisher-id'`)

Appendix P

`self` Axis Example



```
/
  article article-type="research-article"
    front
      journal-meta
        journal-id journal-id-type="nlm-ta"
          Front Zool
        journal-title
          Frontiers in Zoology
        issn pub-type="epub"
          1742-9994
        publisher
          publisher-name
            BioMed Central
          publisher-loc
            London
      article-meta
        article-id pub-id-type="publisher-id"
          1742-9994-3-18
        article-id pub-id-type="pmid"
          17112384
        article-id pub-id-type="doi"
          10.1186/1742-9994-3-18
        article-categories
          subj-group subj-group-type="heading"
            subject
              Methodology
        title-group
          article-title
            Music notation: a new method for visualizing social interaction in animals and humans
        contrib-group
          contrib id="A1" corresp="yes" contrib-type="author"
            name
              surname
                Chase
              given-names
                Ivan D
            xref ref-type="aff" rid="I1"
              1
            xref ref-type="aff" rid="I2"
              2
            email
              ichase@notes.cc.sunysb.edu
        aff id="I1"
          label
            1
          Department of Sociology, Stony Brook University, Stony Brook, NY 11794-4345, USA
        aff id="I2"
          label
            2
          Graduate Program in Ecology and Evolution, Stony Brook University, Stony Brook, NY 11794-5245, USA
        <?more?>
      <?more?>
    <?more?>
```

starting context:
`/descendant::article-categories`

○ context node

✗ node(s) selected

evaluating
`self::node()`

Axis specifier	Node set
`self::`	`article-categories`

Sample XPath Select Expressions for Practice

Let’s Review the Basics

Here are a few simple XPath expressions, that could be used in a @select expression, that is, in relationship to a context node, not used to set context or match a pattern.

`sec/title`	All the <title> children of the <sec> children of the context node
`sec//title`	All the <title> children of all of the node children of the <sec> children of the context node
`../title`	<title> children of the parent of the context node
`@*`	All the attributes of the context node
`/descendant::title[1]`	The first <title >descendent of the root
`//title[1]`	All <title> descendents of the root that are the first <title> child of their parent
`name | collab | email`	The union of the <name> and <collab> and <email> children of the context node
`./title`	<title> children of the context node. (This could also be written as ‘child::title’ or as ‘title’.)
`//list[ancestor::list]`	All <list> elements in the document that have a <list> ancestor
`table-wrap/caption`	All <caption> children of the <table-wrap> children of the context node
`table-wrap[caption]`	All <table-wrap> children of the context node, if and only if they have a <caption> child
`sec[title='Acknowledgements']`	All <sec> children of the context node that have a <title> child whose text value is ‘Acknowledgements’
`sec[contains(title, ‘Acknowledgements']`	All <sec> children of the context node that have a <title> child whose text value contains the string ‘Acknowledgements’
`following-sibling::*[1]`	First following sibling element of the context node
`preceding-sibling::*[1]`	Most recent following sibling element (reverse axis) of the context node
`count(descendant::span)`	The number of <span> element descendants of the context node
`sec[label and title]`	All the <sec> element children of the context node that have *both* <label> and <title> element children
`sec[@sec-type='chapter']`	All the <sec> element children of the context node that have a @sec-type attribute with a value of “chapter”

Now with a Little More Context

`//caption[count(*) > 1 or not(p)]`	All the captions in the document that have more than one child element or do not have a <p> child element
`//sec[title | p]`	All <sec> elements in the document if they have either a <title> or a <p> child
`//graphic[starts-with(@href,'http://')]`	All <graphic> elements in the document, that have an @href attribute that begins with the string “http://”.
`sec[@sec-type='chapter']/title`	The <title> children of the <sec> children of the context node, for the <sec> elements that have a @sec-type attribute with the value ‘chapter’
`item[not(following-sibling::item)]`	<item> children of the context node that do NOT have a following sibling that is also an <item>
`"//xref[@rid = current()/@id]"`	All the <xref> elements in the document for which the @rid attribute of the <xref> is the same as the @id attribute of the context (here the current) element
`sec[@sec-type='intro']//title`	All the <title> descendents (whether section titles, figure title, table title, whatever) of all the node children of the <sec> child of the context node that has a @sec-type attribute with a value of “intro”
`count="table-wrap | table[not(ancestor::tablewrap)]"`	Count the number of <table-wrap> elements plus the number of <table> elements that do not appear inside a <table-wrap
`../label | self::title`	The <label> child of the parent of the context node, or the context node, if that node is a <title> element.
`not(position()=1) and position()=last()`	The context node is the last among its siblings and not also the first among them.
`//xref[@rid = $id]`	All the <xref> elements in the document, for which the @rid attribute of the <xref> equals a variable named $id
`<xsl:variable name="published" select="/article/front/article-meta/issue and not(/article/@preview='yes')"/>`	If there is an <issue> element in the article metadata, and the <article> has no @preview attribute of ‘yes’, then the $published variable is set to true [true()].