

Applied Databases

Lecture 18

XPath and XSLT

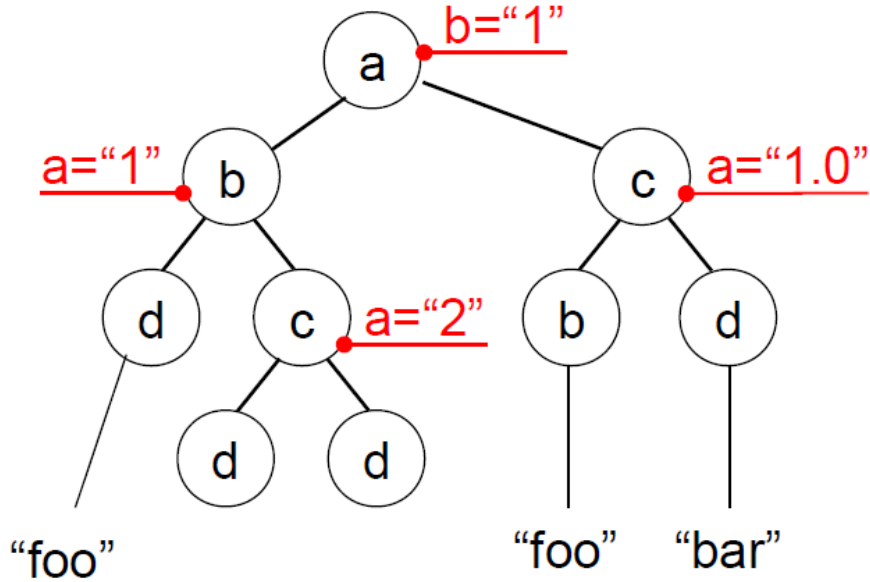
Sebastian Maneth

University of Edinburgh - March 17th, 2016

Outline

1. XPath
2. XSLT – eXtensible Stylesheet Language Transformations

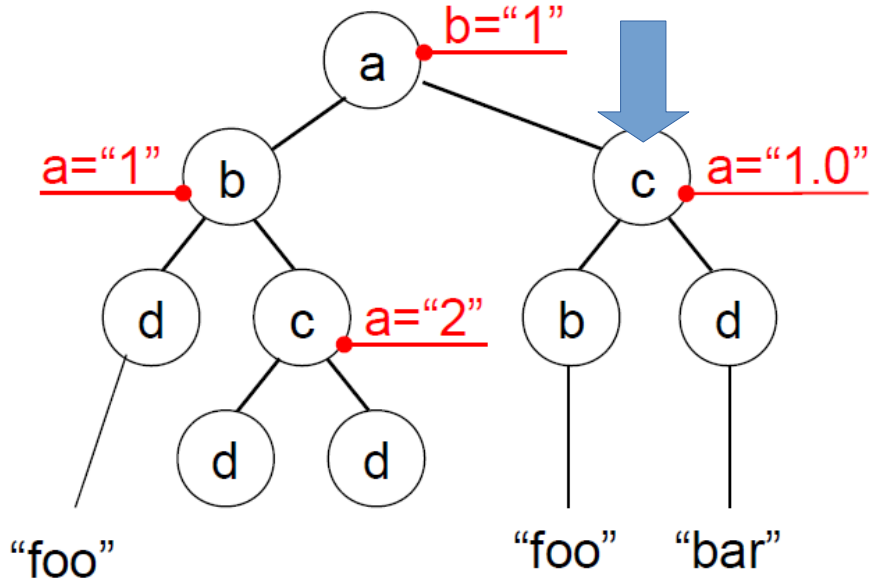
1. XPath



`<a/>`
abbreviation for
`<a>`

Query that selects `c-nodes` that have a `b-child`?

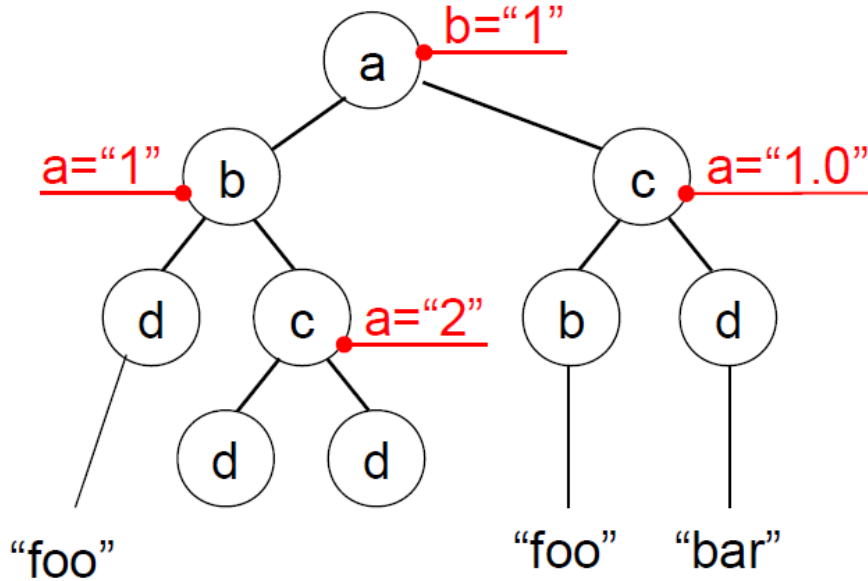
1. XPath



`<a/>`
 abbreviation for
`<a>`

Query that selects `c`-nodes that have a `b`-child? `//c[b]`

XPath

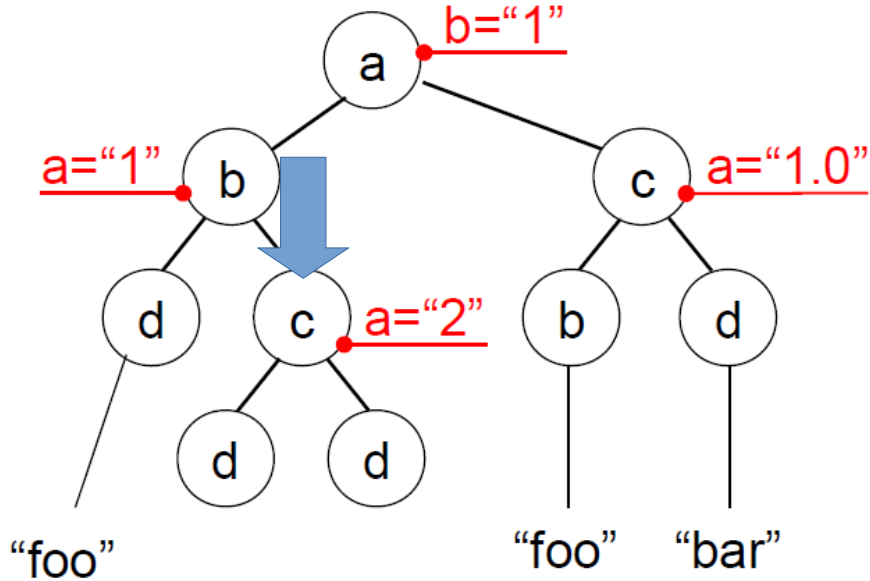


`<a/>`
 abbreviation for
`<a>`

Query that selects c-nodes that have a b-child? `//c[b]`

c-nodes that do NOT have a b-child?

XPath

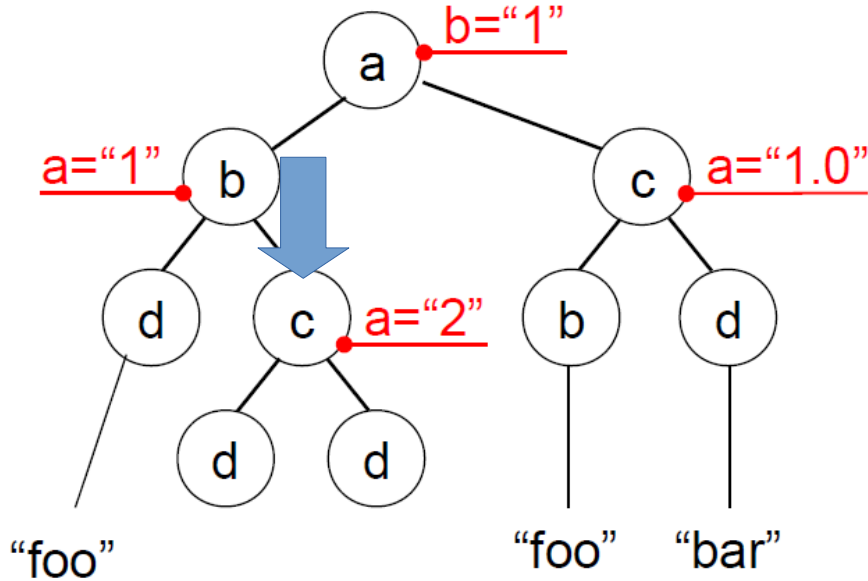


<a/>
 abbreviation for
 <a>

Query that selects c-nodes that have a b-child? //c[b]

c-nodes that do NOT have a b-child? //c[not(b)]

XPath



<a/>
 abbreviation for
 <a>

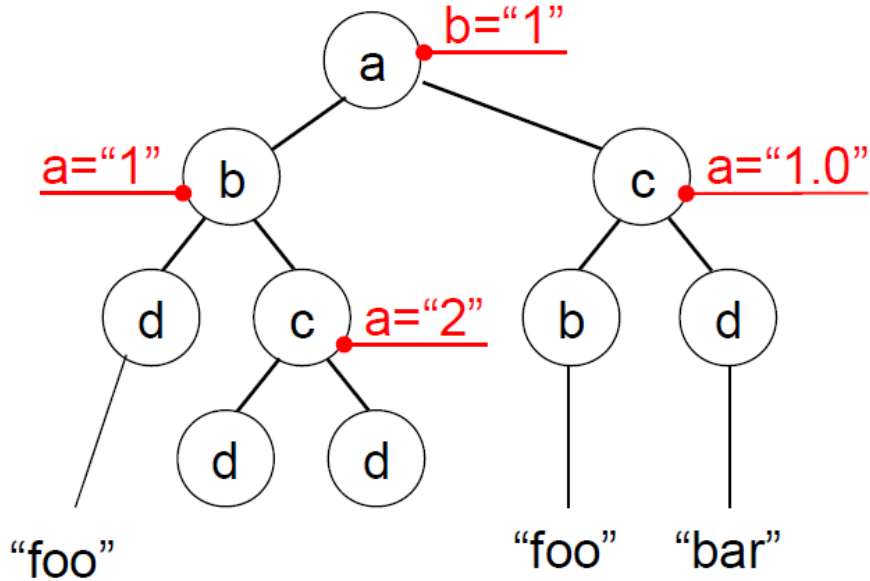
Query that selects c-nodes that have a b-child? `//c[b]`

c-nodes that do NOT have a b-child? `//c[not(b)]`

NEGATE

does there exist a b-node?

XPath

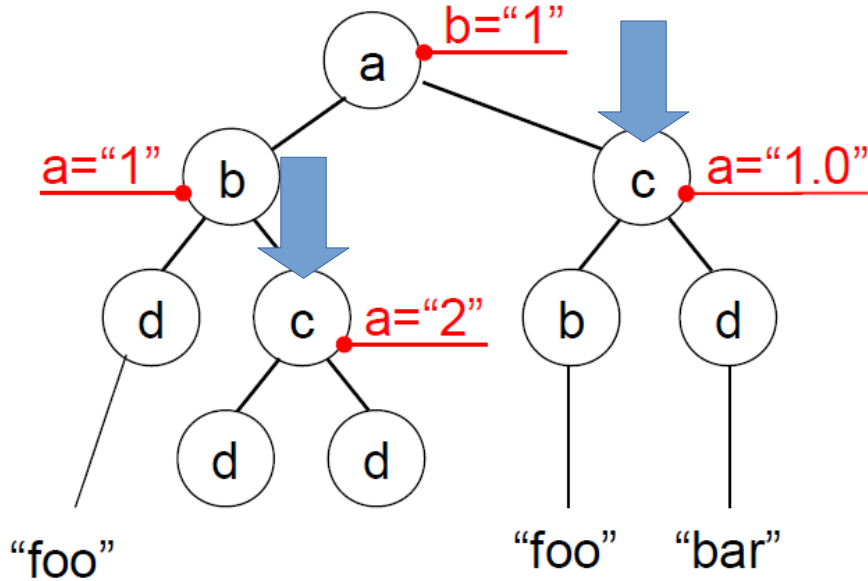


<a/>
 abbreviation for
 <a>

Query that selects c-nodes that have a b-child? //c[b]

c-nodes that have a child not labeled b?

XPath

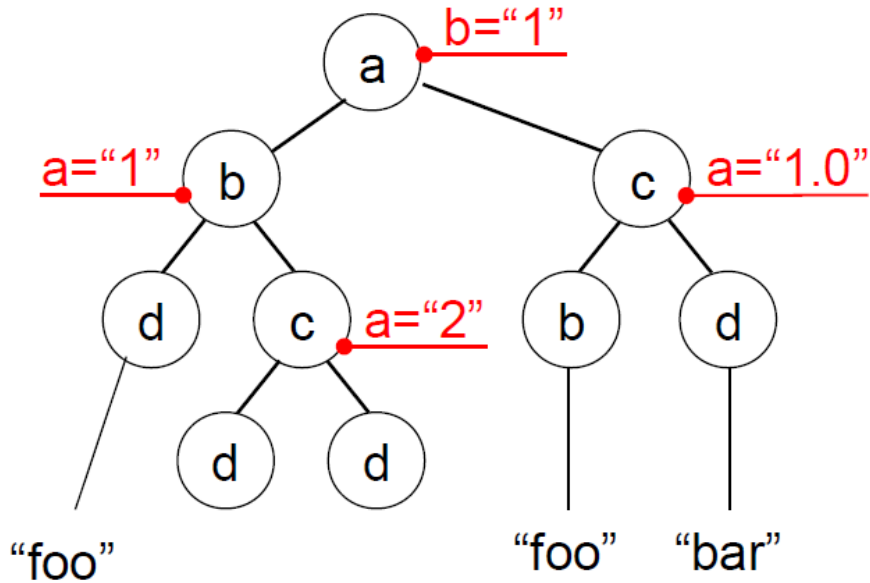


<a/>
 abbreviation for
 <a>

Query that selects c-nodes that have a b-child? `//c[b]`

c-nodes that have a child not labeled b? `//c[*[not(self::b)]]`

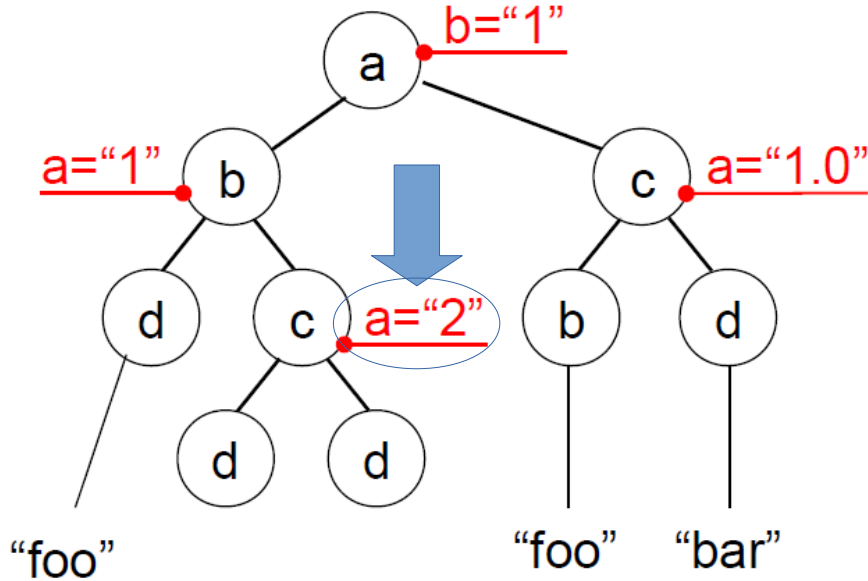
XPath



`<a/>`
abbreviation for
`<a>`

Query that selects **a-attributes with maximal value?**

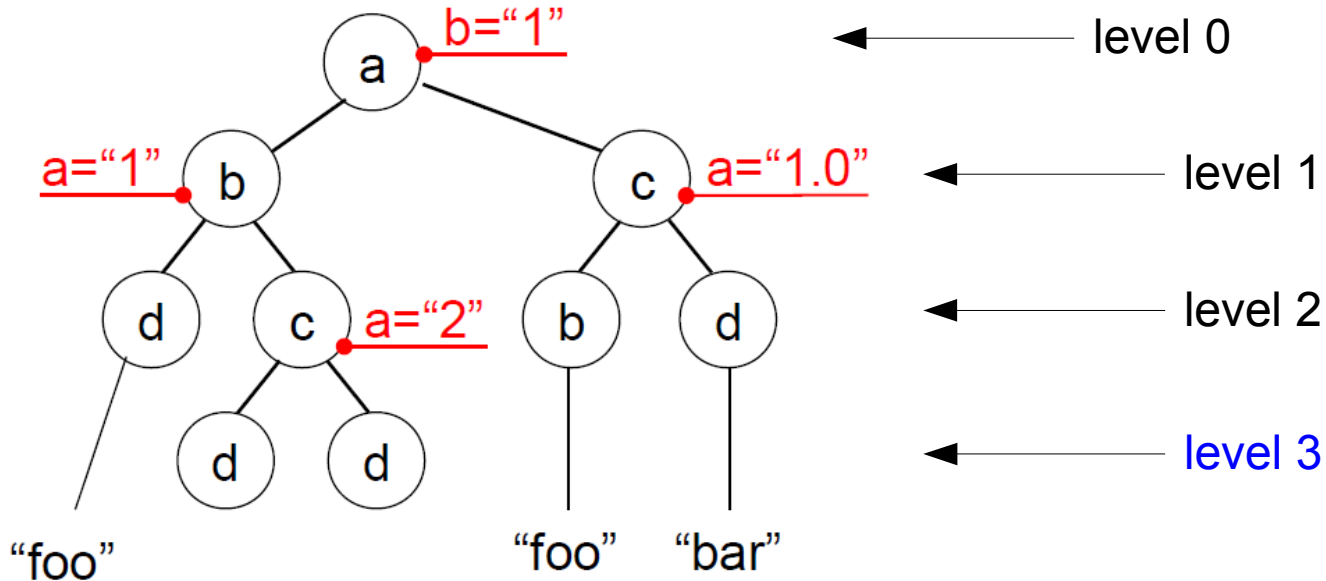
XPath



<a/>
 abbreviation for
 <a>

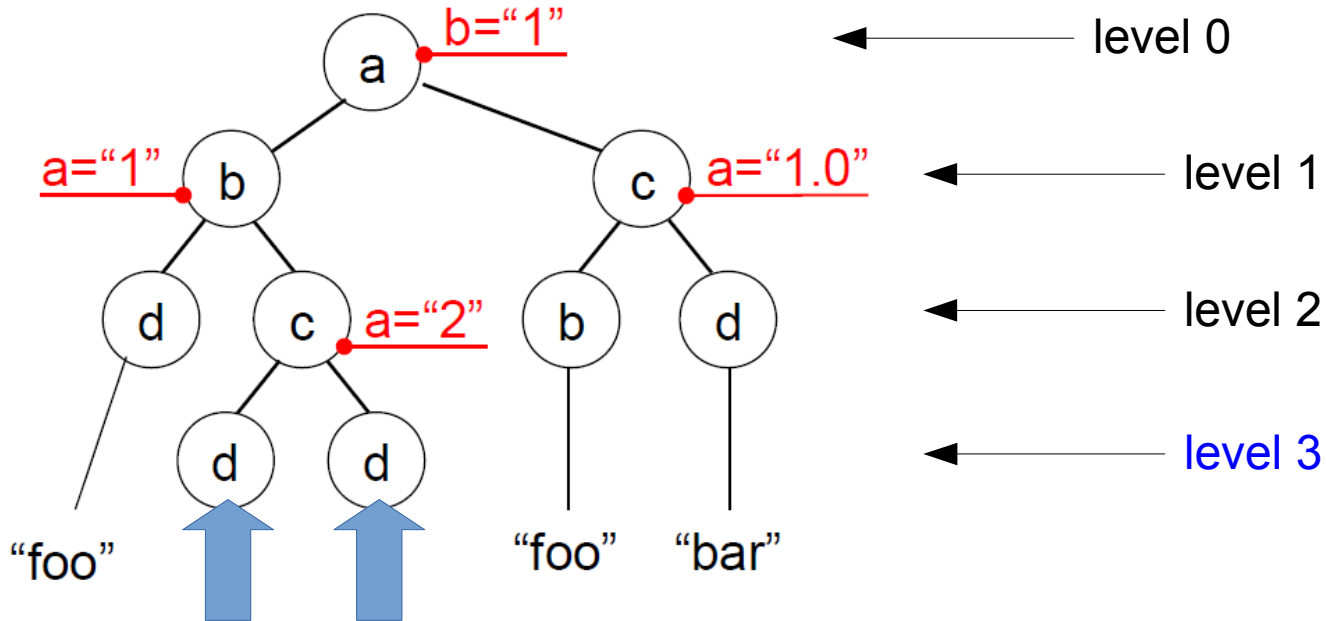
Query that selects **a-attributes** with maximal value? `//@a[not(.< //@a)]`

XPath



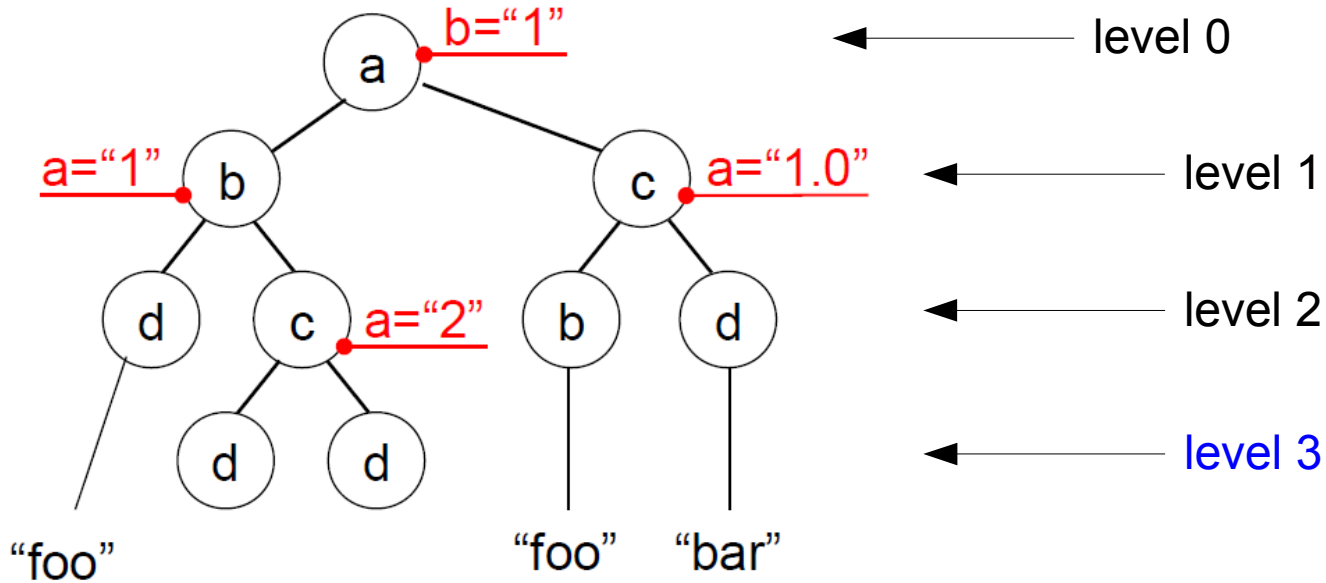
Query that selects **element nodes at level 3 of the tree?**

XPath



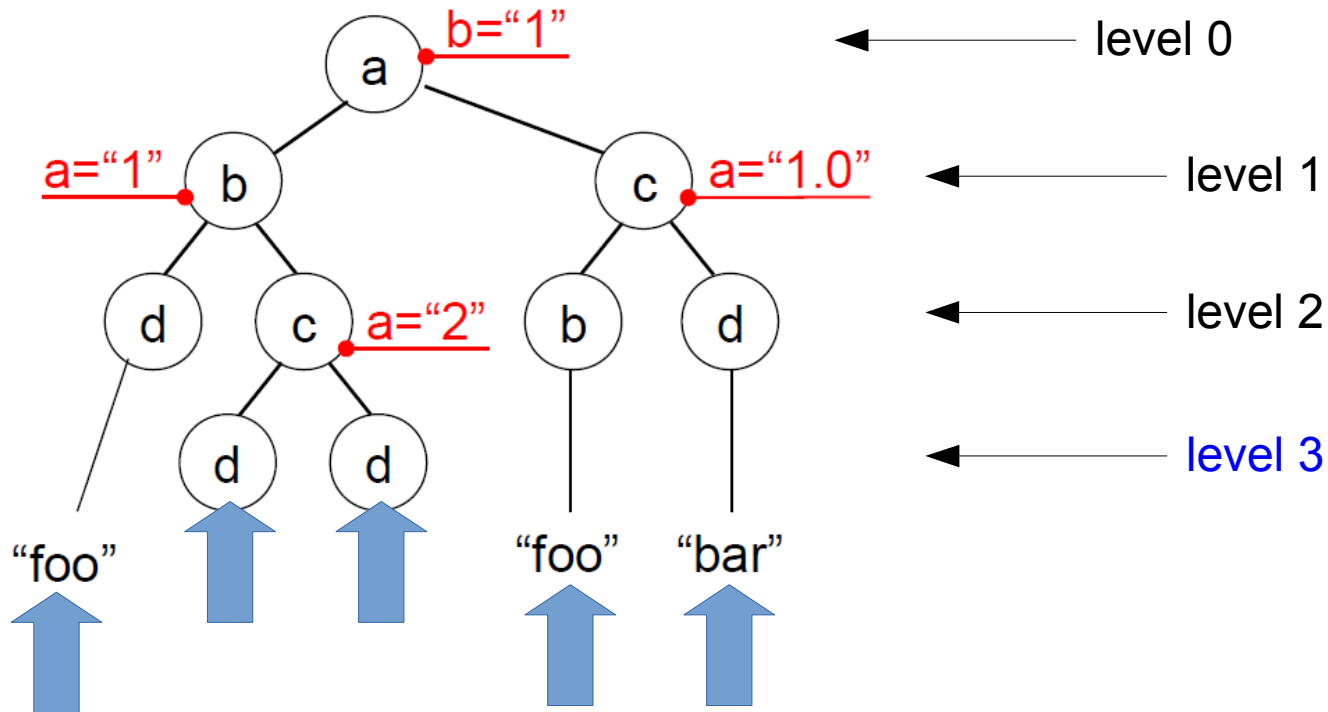
Query that selects **element nodes** at level 3 of the tree? `//*[@*/*/*/*]`

XPath



Query that selects **element and text nodes at level 3 of the tree?**

XPath



Query that selects **element and text nodes at level 3 of the tree?**

```
//*[@*/child::node()]
```

```
//node()[count(ancestor::*=3)]
```

XPATH Tester/Evaluator/Query

An XPATH Tester Tool which runs an XPATH statement against an XML fragment

XPATH Statment

```
//node()[count(ancestor::*)=3]
```

Run XPATH

XML

```
<a b="1"><b a="1"><d>foo</d><c a="2"><d/><d/></c></b><c  
a="1.0"><b>foo</b><d>bar</d></c></a>
```

Result

```
foo  
-----  
<d/>  
-----  
<d/>  
-----  
foo  
-----  
bar
```


Location Steps & Paths

→ A Location Path is a sequence of Location Steps

→ A Location Step is of the form

axis :: **nodetest** [**Filter_1**] [**Filter_2**] ... [**Filter_n**]

Filters (aka predicates, (filter) expressions)
evaluate to **true/false**

nodetest: * or **node-name** (could be expanded → namespaces) or

- **text()**
- **comment()**
- **processing**
-**instruction(ln)**
- **node()**

12 Axes

Forward Axes:

- **self**
- **child**
- **descendant-or-self**
- **descendant**
- **following**
- **following-sibling**

Backward Axes:

- **parent**
- **ancestor**
- **ancestor-or-self**
- **preceding**
- **preceding-sibling**

→ **attribute**

reverse doc order

In doc order

Location Path Semantics

→ A Location Path **P** is a sequence of Location Steps

a_1 :: **n_1** [**F_1_1**] [**F_1_2**] ... [**F_1_n1**]
/ **a_2** :: **n_2** [**F_2_1**] [**F_2_2**] ... [**F_2_n2**]

/ **a_m** :: **n_m** [**F_m_1**] [**F_m_2**] ... [**F_m_nm**]

S0 = initial sequence of context-nodes

- (1) (to each) context-node **N** in **S0**, apply axis **a_1**: gives sequence **S1** of nodes
- (2) remove from **S1** any node **M** for which
 - test **n_1** evaluates to false
 - any of filters **F_1_1**, ..., **F_1_n1** evaluate to false.

Proceed similarly for **S1** and **a_2**, et cetera

Finally, obtain **Sm** = result sequence of query **P**.

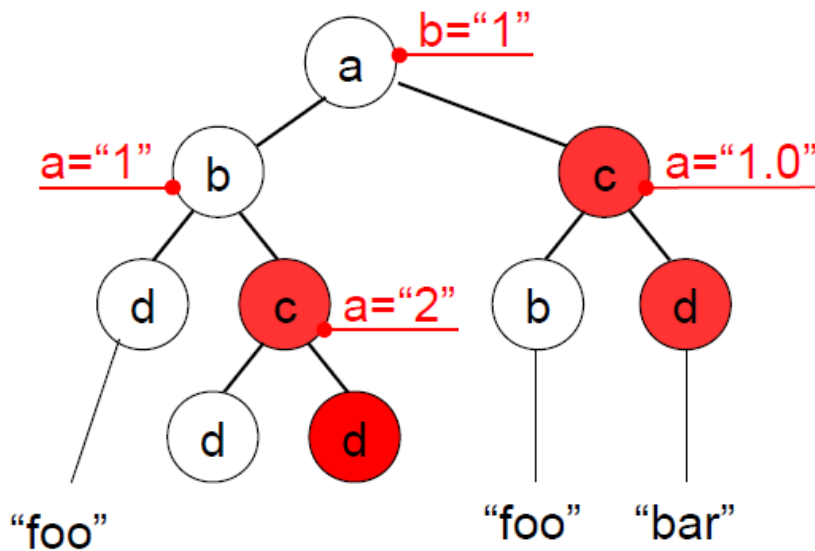
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



`//*[position()=2]`

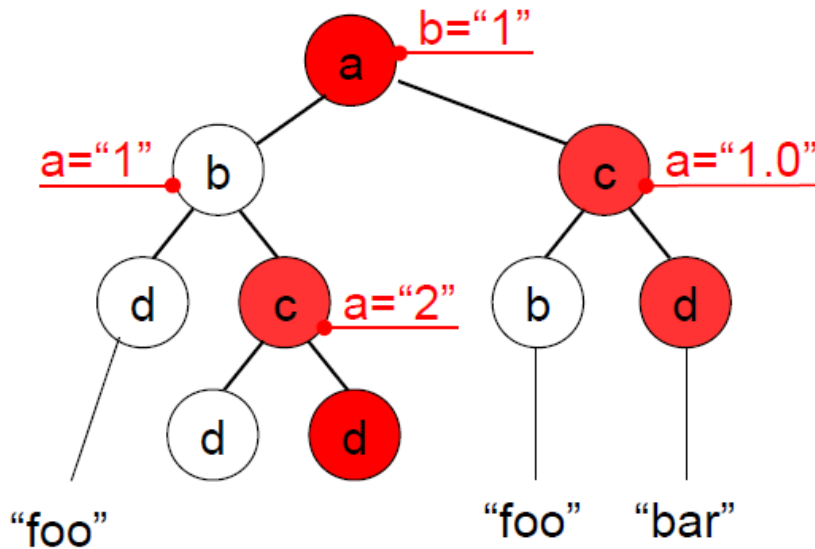
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
/**[position()=2]
```

```
/**[position()=last()]
```

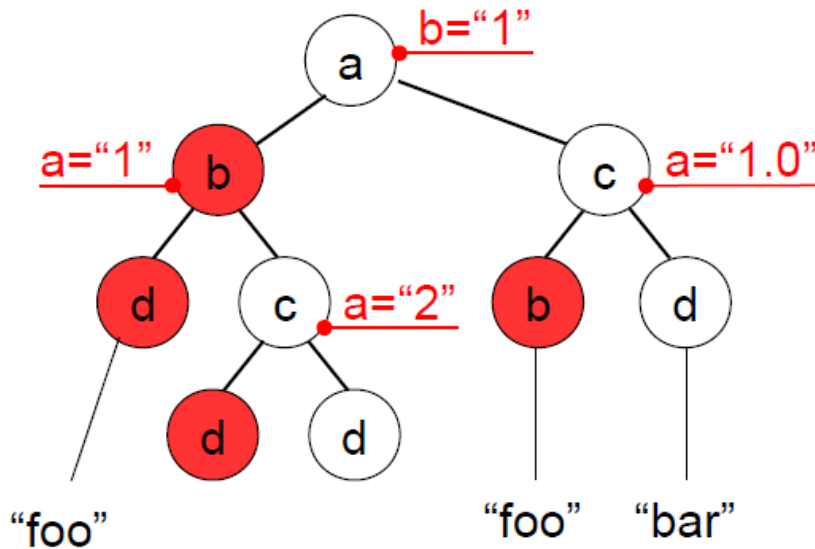
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
/**[position()=2]
```

```
/**[position()=last()-1]
```

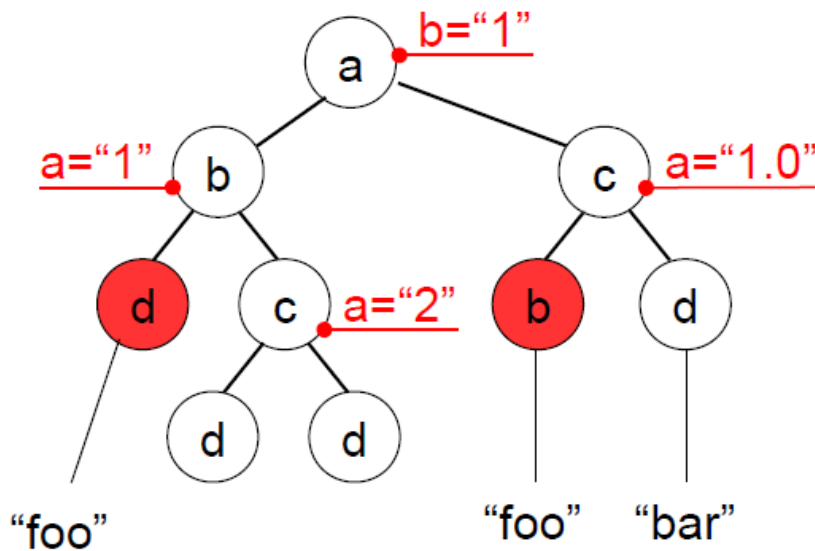
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
/**[position()=2]
```

```
/**[position()=last()-1  
and ./text()="foo"]
```

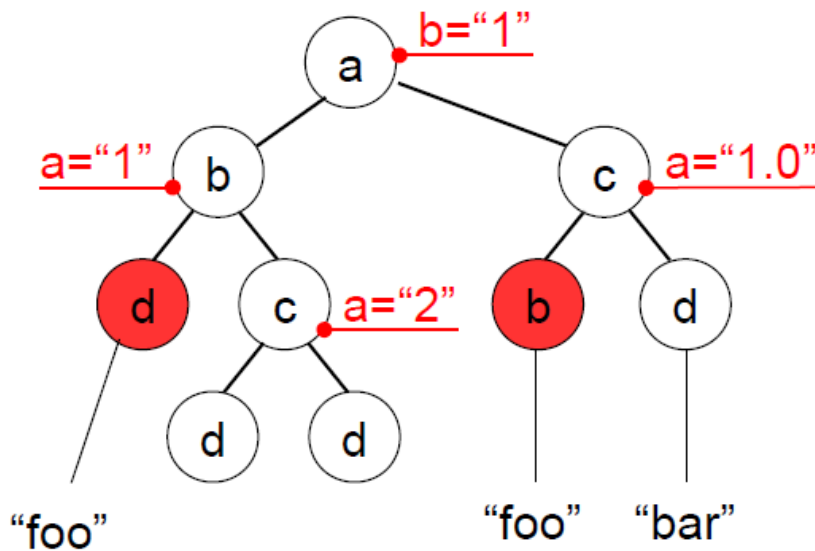
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



```
//*[position()=2]
```

```
//*[position()=last()-1  
and ./text()="foo"]
```

Useful:

```
child::*[self::chapter or self::appendix][position()=last()]
```

selects the last chapter or appendix child of the context node

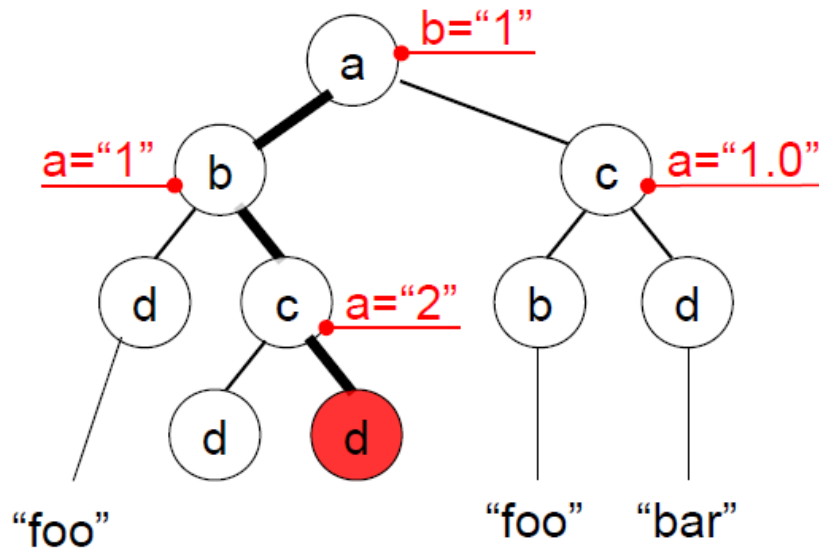
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



Melvil Dewey

`/**[position()=1]/**[position()=2]/**[position()=2]`

→ allows absolute location of any node (a la Dewey)

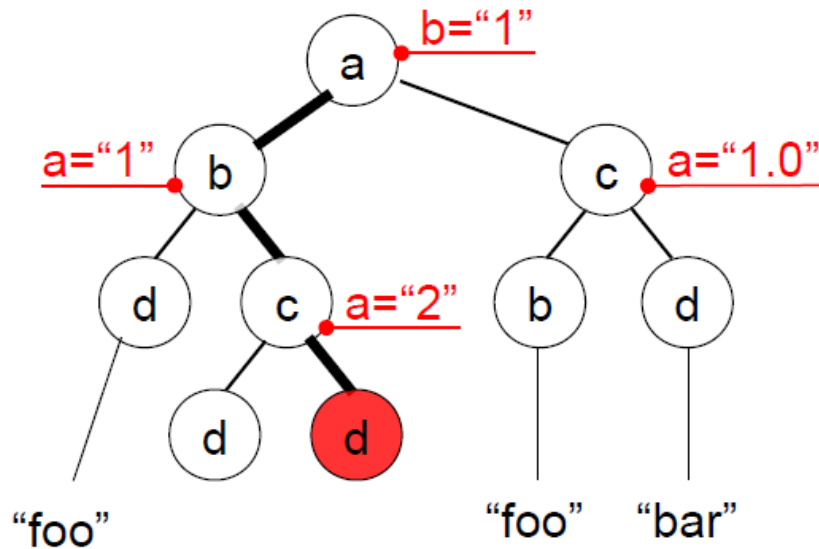
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



Melvil Dewey

`/**[position()=1]/**[position()=2]/**[position()=2]`

Abbreviation: `/**[1]/**[2]/**[2]`

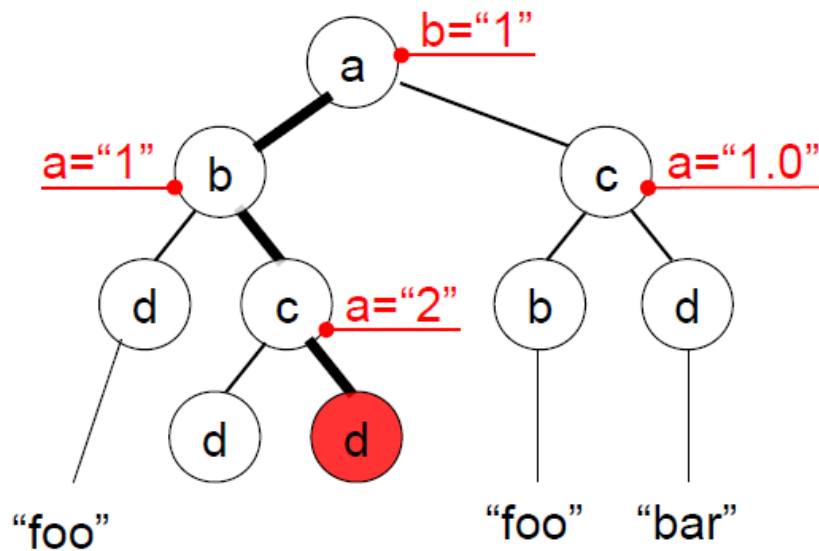
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



`/**[position()=1]/**[position()=2]/**[position()=2]`

Abbreviation: `/**[1]/**[2]/**[2]` → What is result for `/**[./**[2]/**[2]]`

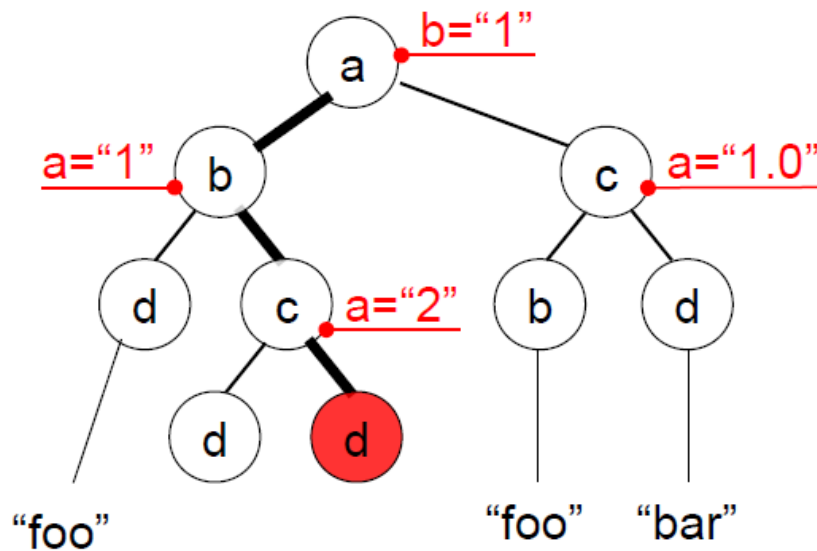
Useful Functions (on Node Sets)

→ `last()`

returns context-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



What is the result for
`//d[position() = last()]`

`//*[position()=1]//*[position()=2]//*[position()=2]`

Abbreviation: `//*[1]//*[2]//*[2]` → **What is result for** `//*[.//*[2]//*[2]]`

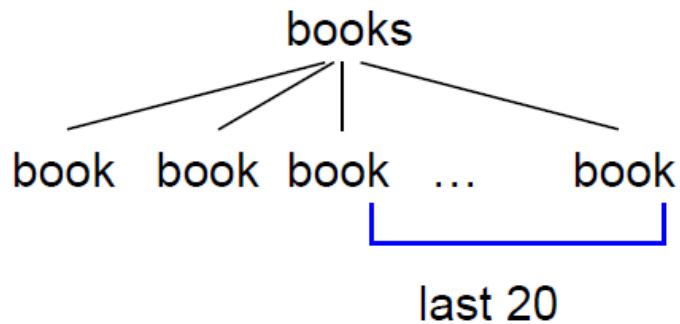
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



How do you select the
last 20 book-children of books?

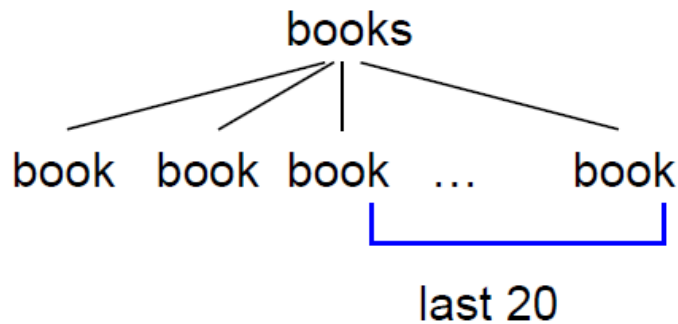
Useful Functions (on Node Sets)

→ `last()`

returns contex-size from the evaluation context

→ `position()`

Returns context-position from the eval. context



How do you select the
last 20 book-children of books?

`/books/book[position()>last()-20]`

XPath Evaluation

- can be done polynomial time
- based on “context-value tables”
- careful recording of unique values (no duplicates)

2. XSLT

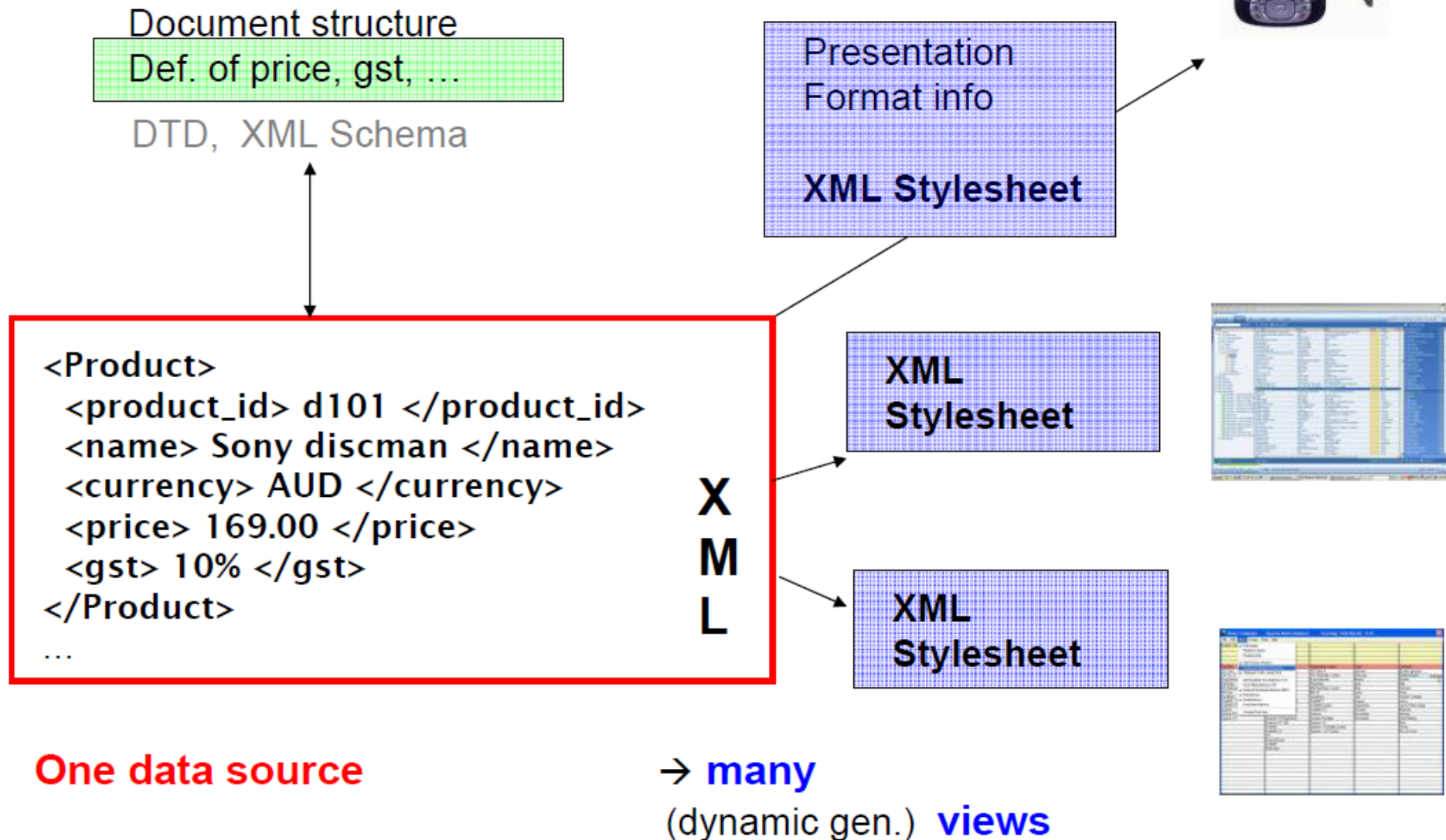
2. XSLT

- W3C Standard
- developed early on (XSL in 1998)
- Version 2.0 (W3C recommendation 2007)
- Version 3.0 (streaming)
W3C candidate recommendation (15. November 2015)

All major browsers (Chrome, Firefox, IE, Safari, etc)
support XSLT

- if an XML document has an XSL stylesheet associated with it,
then browser transforms it on-the-fly

XML, typical usage scenario



XML

- General data format
 - Includes no formatting information
-

Usage Idea

- Store data *once*, in most general format
(free tech writers from bothering with layout issues)
- Reuse fragments of the data (same contents; looks different depending on the context)

E.g. different output formats (size, device)
style tailored to reader's preference
style tailored to adhere w. corporate/web site identity

eXtensible Stylesheet Language

“stylesheet” = recipe how to *display* your XML data

e.g. “display titles of books in bold, large font”

“display authors of books in italic, medium size”

etc.

→ Choose an output format (e.g. XHTML, HTML, text, PDF, etc.)

→ Transform the XML:

- add formatting information
- rearrange data
- sorting
- delete certain parts, copy others

etc

Example Transformations

- XML to **HTML** – for browser display
- XML to **LaTeX** – for TeX layout
- XML to **SVG** – graphs, charts, trees
- XML to **tab-delimited** – for DB/stat packages
- XML to **plain text, e.g., PDF** – for printing



e.g. print bills for a telecom company

→ data comes in XML

→ want to produce bills in PDF

Example Transformations

- XML to **HTML** – for browser display
- XML to **LaTeX** – for TeX layout
- XML to **SVG** – graphs, charts, trees
- XML to **tab-delimited** – for DB/stat packages
- XML to **plain text, e.g., PDF** – for printing


e.g. print bills for a telecom company
→ data comes in XML
→ want to produce bills in PDF

XSLT vs. custom **SAX/DOM traversal** (as your XML→CSV from Assignment 1)

- smaller code
- better readability & maintainability
- let XSLT compiler attempt to *stream* the transformation

Example Transformations

- smaller code
- better readability & maintainability
- let XSLT compiler attempt to **stream** the transformation

- 
- “best effort” approach
 - very hard to detect statically
(interesting research question)
 - XSLT 3.0

Some academic transformation languages even have

- static type checking
- compiler can verify if all outputs are, e.g., correct XHTML



XSLT program: list of **template** (rules) – written in XML syntax
Template: pattern → action

Template

```
<xsl:template match="para">  
  <p><xsl:apply-templates/></p>  
</xsl:template>
```

(relative) XPath expression
→ current node labeled “**para**”?

output p-node (in XML)

recursively descent & apply templates

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

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

</xsl:stylesheet>
```

para.xslt



XSLT program

- changes "para" nodes to "p"-nodes
- deletes all other element nodes

```
$ cat t1.xml
<x><t/><para><para/></para></x>

$ xsltproc para.xslt t1.xml
<?xml version="1.0"?>
<p><p/></p>
```



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

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

</xsl:stylesheet>
```

para.xslt



XSLT program

- changes “para” nodes to “p”-nodes
- deletes all other element nodes
- **keeps all text-nodes!**

```
$ cat t2.xml
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

$ xsltproc para.xslt t2.xml
<?xml version="1.0"?>
this is a <p>hello world</p>
```

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

<xsl:template match="node()">
  <xsl:copy><xsl:apply-templates/></xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

trans.xslt

```
$ cat t2.xml
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

$ xsltproc trans.xslt t2.xml
<?xml version="1.0"?>
<a>this<b>is a <d/><e/></b><para>hello world<c/></para></a>
```

- all nodes are matched and taken over
- **except** *attributes nodes* (they are deleted)

```

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="node() | @*">
    <xsl:copy><xsl:apply-templates select="node() | @*" /></xsl:copy>
  </xsl:template>
</xsl:stylesheet>

```

identity.xslt

```

$ cat t2.xml
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

$ xsltproc identity.xslt t2.xml
<?xml version="1.0"?>
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

```

→ all nodes are matched and taken over

```

<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="node() | @*">
    <xsl:copy><xsl:apply-templates select="node() | @*" /></xsl:copy>
  </xsl:template>
</xsl:stylesheet>

```

identity.xslt

```

$ cat t2.xml
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

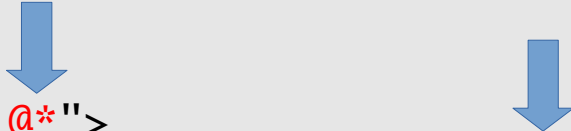
$ xsltproc identity.xslt t2.xml
<?xml version="1.0"?>
<a z="TT">this<b>is a <d a="1278"/><e/></b>
<para u="18">hello world<c/></para></a>

```

- all nodes are matched and taken over
- compare the (size of the) above XSLT program to SAX program for the same task!

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="node() | @*">
    <xsl:copy><xsl:apply-templates select="node() | @*" /></xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

identity.xslt



- all nodes are matched and taken over
- compare the (size of the) above XSLT program to SAX program for the same task!

- would be interesting to rewrite the ebay XML → CSV converter in XSLT

XPATH Tester/Evaluator/Query

An XPATH Tester Tool which runs an XPATH statement against an XML fragment

XPATH Statement

Run XPATH


XML

```
<a b="1"><b a="1"><d>foo</d><c a="2"><d/><d/></c></b><c  
a="1.0"><b>foo</b><d>bar</d></c></a>
```

Result

```
<a b="1"><b a="1"><d>foo</d><c a="2"><d/><d/></c></b><c  
a="1.0"><b>foo</b><d>bar</d></c></a>  
-----  
b="1"  
-----  
<b a="1"><d>foo</d><c a="2"><d/><d/></c></b>  
-----  
a="1"  
<d>foo</d>  
-----  
foo  
-----  
<c a="2"><d/><d/></c>  
-----  
a="2"  
-----  
<d/>  
-----  
<d/>  
-----  
<c a="1.0"><b>foo</b><d>bar</d></c>  
-----  
a="1.0"  
-----  
<b>foo</b>  
-----  
foo  
-----  
<d>bar</d>  
-----  
bar
```

```
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="node() | @*">
    <xsl:copy><xsl:apply-templates select="."/;></xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```



What will this program do?

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



```
<xsl:template match="node() | @*">
  <xsl:copy><xsl:apply-templates select="."/></xsl:copy>
</xsl:template>
```

```
</xsl:stylesheet>
```

```
$ xsltproc identity.xslt t2.xml
```

```
runtime error: file loop.xslt line 6 element copy
```

```
xsltApplyXSLTTemplate: A potential infinite template recursion was
detected.
```

You can adjust `xsltMaxDepth` (`--maxdepth`) in order to raise the maximum number of nested template calls and variables/params (currently set to 3000).

Templates:

```
#0 name node() | @*
```

```
#1 name node() | @*
```

```
#2 name node() | @*
```



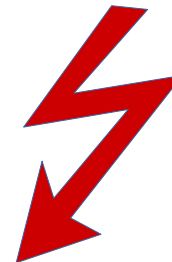
```
<xsl:stylesheet  
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
version="1.0">
```

```
<xsl:template match="node() | @*">  
  <xsl:copy><xsl:apply-templates select="."/></xsl:copy>  
</xsl:template>
```

```
</xsl:stylesheet>
```



- XSLT is Turing-complete
- easy to write non-terminating programs



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

<xsl:template match="node() | @*">
  <xsl:copy><xsl:apply-templates select="."/></xsl:copy>
</xsl:template>

<xsl:template match="b">
  <fff><xsl:apply-templates select="."/></fff>
</xsl:template>

</xsl:stylesheet>
```

→ both templates are applicable to **b-nodes**

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

<xsl:template match="node() | @*">
  <xsl:copy><xsl:apply-templates select="."/></xsl:copy>
</xsl:template>

<xsl:template match="b">
  <fff><xsl:apply-templates select="."/></fff>
</xsl:template>

</xsl:stylesheet>
```

- both templates are applicable to **b-nodes**
- more specific template is used (“**b**” before “**node()**”)

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

<xsl:template match="node() | @*">
  <xsl:copy><xsl:apply-templates select="."/></xsl:copy>
</xsl:template>

<xsl:template match="b">
  <fff><xsl:apply-templates select="."/></fff>
</xsl:template>

</xsl:stylesheet>
```

```
$ xsltproc relab.xslt t2.xml
<?xml version="1.0"?>
<a z="TT">this<fff>is a <d a="1278"/><e/></fff><para u="18">hello
world<c/></para></a>
```

Conflict Resolution and Modes in XSLT

- Note that for each node visited by the XSLT processor (cf. default template ②), **more than one template might yield a match.**
- XSLT assigns a **priority** to each template. The more specific the template pattern, the higher the priority:

```
<xsl:template match="e"> cons </xsl:template>
```

Pattern <i>e</i>	Priority
*	-0.5
<i>ns</i> :*	-0.25
element/attribute name	0
any other XPath expression	0.5

- **Example:**

Priority of `author` is 0, priority of `/strip/prolog/author` is 0.5.

- Alternatively, make priority explicit: any real number

```
<xsl:template priority="p" ...>
```

Delete all nested <list>s

```
<xsl:template match="list/list"  
  priority="2">  
  <!-- deleted nested list -->  
</xsl:template>
```

```
<xsl:template match="list"  
  priority="1">  
  <list><xsl:apply-templates/></list>  
</xsl:template>
```

```
<d>  
<list>text1  
<list>  
blah  
</list>  
</list>  
<list>  
next  
</list>  
</d>
```



```
<?xml version="1.0"?>  
<list>text1  
</list>  
<list>  
next  
</list>
```

Delete all nested <list>s

```
<xsl:template match="list/list"
  >
  <!-- deleted nested list -->
</xsl:template>
```

```
<xsl:template match="list"
  >
  <list><xsl:apply-templates/></list>
</xsl:template>
```

```
<d>
<list>text1
</list>
blah
</list>
</list>
<list>
next
</list>
</d>
```



```
<?xml version="1.0"?>
<list>text1
</list>
<list>
next
</list>
```

→ if no priorities are given,
what is the output?

Question

How do you remove `book//author/first` elements, but keep everything else in the XML?

Question

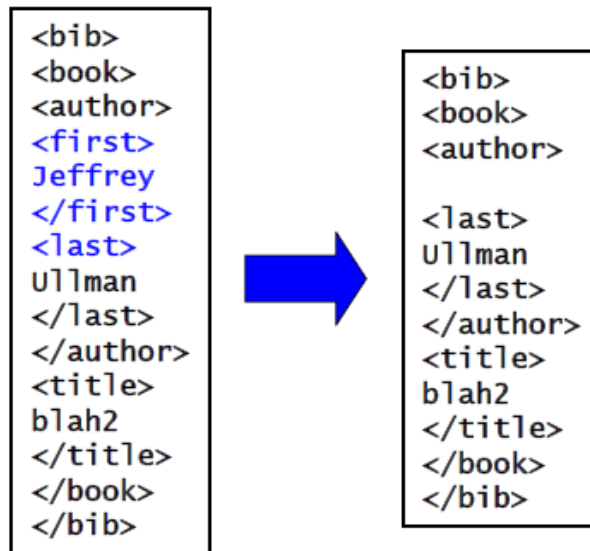
How do you remove `book//author/first` elements, but keep everything else in the XML?

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

<xsl:template match="book//author/first" priority="5">
</xsl:template>

<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```



Context

Quite often, an XSLT stylesheet wants to be **context-aware**.

- Since the XSLT priority mechanism is *not* dynamic, this can cause problems.

Example: Transform the following XML document (sectioned text with cross references) into XHTML:


self-ref.xml

```
1 <section id="intro">
2   <title>Introduction</title>
3   <para> This section is self-referential: <xref to="intro">. </para>
4 </section>
```

We want to generate XHTML code that looks somewhat like this:

self-ref.html

```
1 <h1>Introduction</h1>
2 <p> This section is self-referential: <em>Introduction</em>. </p>
```

 The section title needs to be processed twice, once to produce the heading and once to produce the cross reference.

The “obvious” XSLT stylesheet produces erroneous output:

buggy-self-ref.xsl

```
1 <?xml version="1.0"?>
2 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3     version="1.0">
4
5 <xsl:template match="title">
6     <h1><xsl:apply-templates/></h1>
7 </xsl:template>
8
9 <xsl:template match="para">
10    <p><xsl:apply-templates/></p>
11 </xsl:template>
12
13 <xsl:template match="xref">
14     <xsl:apply-templates select="id(@to)/title"/>
15 </xsl:template>
16
17 </xsl:stylesheet>
```

buggy-output.html

```
1 <h1>Introduction</h1>
2 <p> This section is self-referential: <h1>Introduction</h1>. </p>
```

XSLT modes

- We need to make the processing of the `title` element aware of the context (or **mode**) it is used in: inside an `xref` or not.
- This is a job for **XSLT modes**.
 - ▶ In `<xsl:apply-templates>` switch to a certain mode *m* depending on the context:

```
<xsl:apply-templates mode="m" .../>
```

- ▶ After mode switching, only `<xsl:template>` instructions with a `mode` attribute of value *m* will match:

```
<xsl:template mode="m" .../>
```

- ▶ As soon as `<xsl:apply-templates mode="m" .../>` has finished matching nodes, the previous mode (if any) is restored.

self-ref.xsl


```
1 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
2     version="1.0">  
3  
4 <xsl:template match="title">  
5     <h1><xsl:apply-templates/></h1>  
6 </xsl:template>  
7  
8 <xsl:template match="title" mode="ref">  
9     <em><xsl:apply-templates/></em>  
10 </xsl:template>  
11 .  
12 .  
13 .  
14 <xsl:template match="xref">  
15     <xsl:apply-templates select="id(@to)/title" mode="ref"/>  
16 </xsl:template>  
17  
18 </xsl:stylesheet>
```

output.html

```
1 <h1>Introduction</h1>  
2 <p> This section is self-referential: <em>Introduction</em>. </p>
```

More on XSLT

XSLT Instruction	Effect
<code>xsl:choose</code> , <code>xsl:when</code>	switch statement (ala C)
<code>xsl:call-template</code>	explicitly invoke a (named) template
<code>xsl:for-each</code>	replicate result construction for a sequence of nodes
<code>xsl:import</code>	import instructions from another stylesheet
<code>xsl:output</code>	influence XSLT processor's output behaviour
<code>xsl:variable</code>	set/read variables

- For a complete XSLT reference, refer to  <http://www.w3.org/TR/xslt>
- Apache's Cocoon is an XSLT-enabled web server (see <http://xml.apache.org/cocoon/>).

Syntax

```
<xsl:output  
method="xml|html|text|name"  
version="string"  
encoding="string"  
omit-xml-declaration="yes|no"  
standalone="yes|no"  
doctype-public="string"  
doctype-system="string"  
cdata-section-elements="namelist"  
indent="yes|no"  
media-type="string"/>
```

This example declares a **global variable** para-font-size, which it references in an attribute value template.

```
<xsl:variable name="para-font-size">12pt</xsl:variable>
```

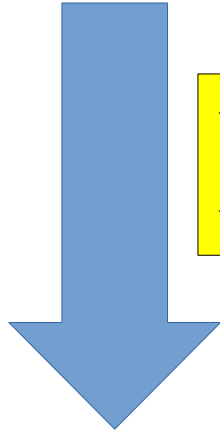
```
.  
.
```

```
<xsl:template match="para">  
  <fo:block font-size="{ $para-font-size }">  
    <xsl:apply-templates/>  
  </fo:block>  
</xsl:template>
```


<xsl:number>

- *Used to generate auto-numbering*
- ```
<xsl:number
 level="single|multiple|any"
 count="pattern" -- which nodes count?
 from="pattern" -- starting point
 value="number-expr" -- force value
 format="s" -- (not covering)
 lang="lg" -- lang to use
 letter-value="alphabetic|traditional"
 grouping-separator="char" -- 1,000
 grouping-size="number" -- 3 in EN
>
```

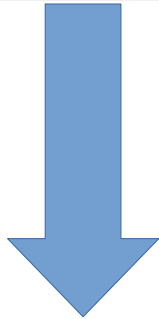
```
<colors>
 <color>red</color>
 <color>green</color>
 <color>blue</color>
 <color>yellow</color>
</colors>
```



```
<xsl:template match="color">
 <xsl:number/>. <xsl:apply-templates/>
</xsl:template>
```

1. red
2. green
3. blue
4. yellow

```
<colors>
 <color>red</color>
 <color>green</color>
 <color>blue
 <color>robin's egg</color>
 <color>navy</color>
 <color>cerulean</color>
 </color>
 <color>yellow</color>
</colors>
```



```
<xsl:template match="color">
 <xsl:number level="multiple" format="1. " />
 <xsl:apply-templates/>
</xsl:template>
```

1. red
2. green
3. blue
  - 3.1. robin's egg
  - 3.2. navy
  - 3.3. cerulean
4. yellow

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

```
<xsl:template match="@*|node()">
 <xsl:copy>
 <xsl:number level="multiple" count="*" />
 <xsl:apply-templates select="@*|node()" />
 </xsl:copy>
</xsl:template>
```

```
</xsl:stylesheet>
```

```
<c>
<c>
<AA>
<a1/>
<a2/>
<a3><u/></a3>
</AA>
</c>

<BB>
<b1/>
<b2/>
<b3/>
</BB>

</c>
```



```
<?xml version="1.0"?>
<c>1
<c>1.1
<AA>1.1.1
<a1>1.1.1.1</a1>
<a2>1.1.1.2</a2>
<a3>1.1.1.3<u>1.1.1.3.1</u></a3>
</AA>
</c>
1.2
<BB>1.2.1
<b1>1.2.1.1</b1>
<b2>1.2.1.2</b2>
<b3>1.2.1.3</b3>
</BB>

</c>
```

**END**

**Lecture 18**