Styling XML

Hans C. Arents

senior IT market analyst

I.T. Works

"Guiding the IT Professional"

Innovation Center, Technologiepark 3, B-9052 Gent (Belgium), Tel: +32 (0)9 241 56 21 - Fax: +32 (0)9 241 56 56

E-mail: hca@itworks.be - Site: http://www.itworks.be/ - Home: http://www.arents.be/

Styling XML

- Part 1: **Transforming XML** 13u30 – 16u00

  coffee break

- Part 2: Displaying XML 16u30 – 17u30

  questions & answers

Transforming XML

Hans C. Arents

senior IT market analyst

I.T. Works

"Guiding the IT Professional"

Innovation Center, Technologiepark 3, B-9052 Gent (Belgium), Tel: +32 (0)9 241 56 21 - Fax: +32 (0)9 241 56 56

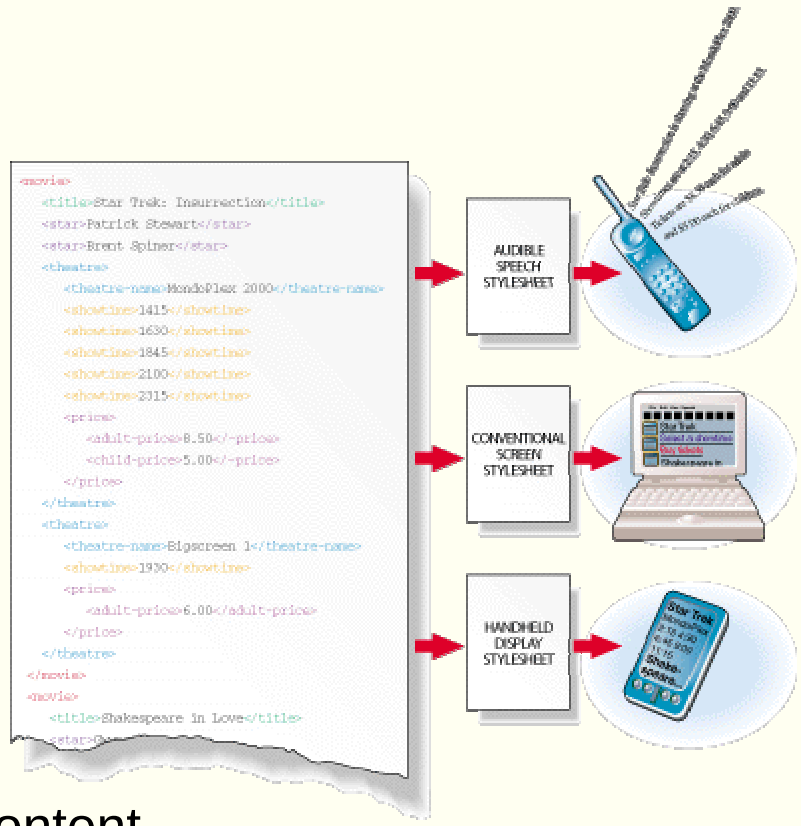
E-mail: hca@itworks.be - Site: http://www.itworks.be/ - Home: http://www.arents.be/

Transforming XML

- Understanding XSL
  - what is XSL?
  - what are the two processes in XSL?
- The XSLT data model
  - how does XSLT see a document?
- The XPath language
  - concepts and syntax
  - how does XPath step through a document?
- The XSLT language
  - concepts and syntax
  - how does XSLT transform a document?
- Tips on XSLT development

XSL: Extensible Stylesheet Language

- XML stylesheet mechanism
  - based on DSSSL
  - compatible with CSS 1.0, 2.0 & 3.0
  - advanced capabilities:
    - contents can be reordered/sorted
    - text can be deleted/duplicated/moved
    - new information can be “computed” from existing information or structures



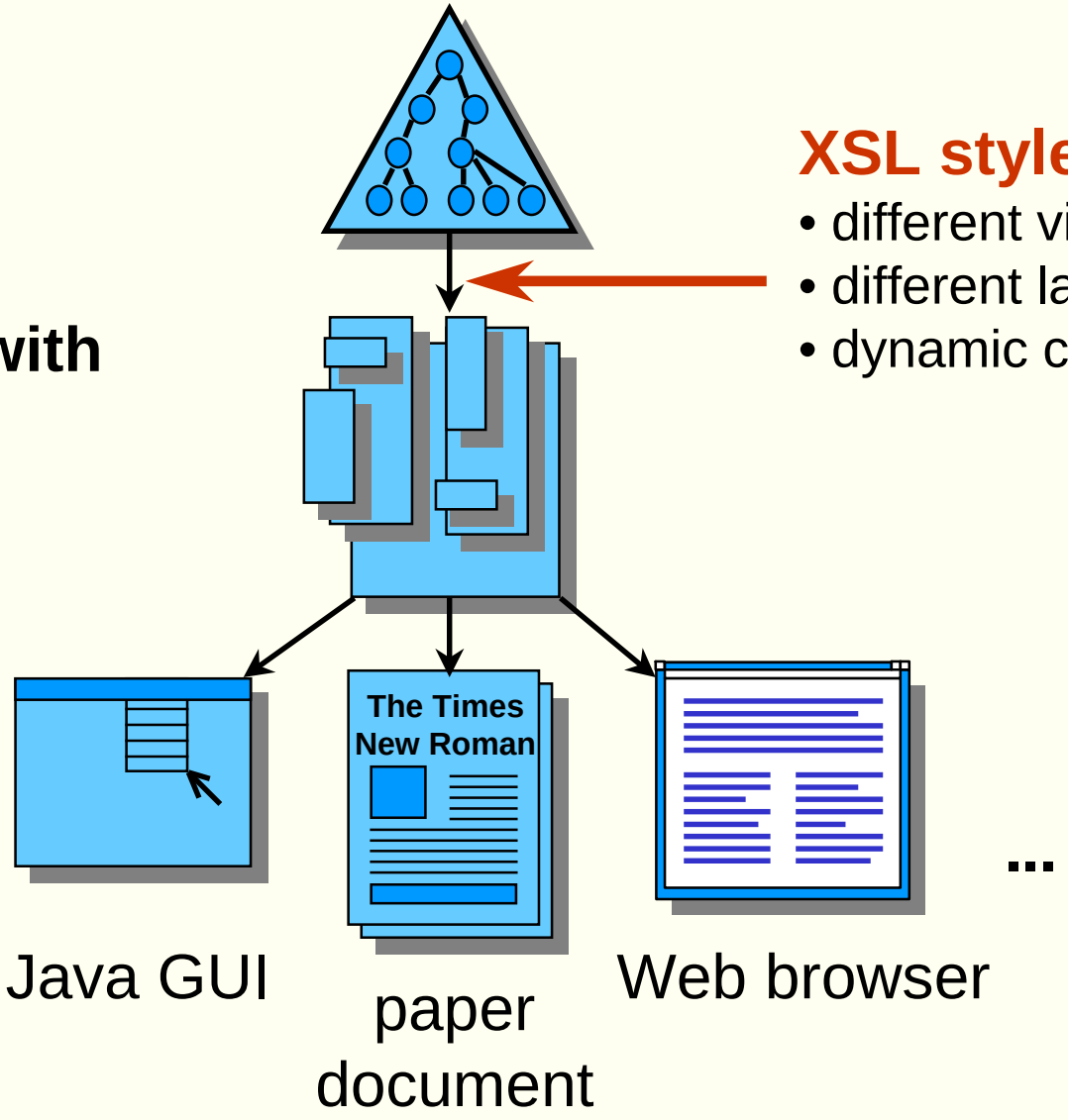
- Useful for:
  - keeping presentation separate from content
  - single source, multiple output media
  - reformatting / re-using content
- Not just document formatting, but also data restructuring!

XSL: Extensible Stylesheet Language

source tree

result tree with formatting objects

output media



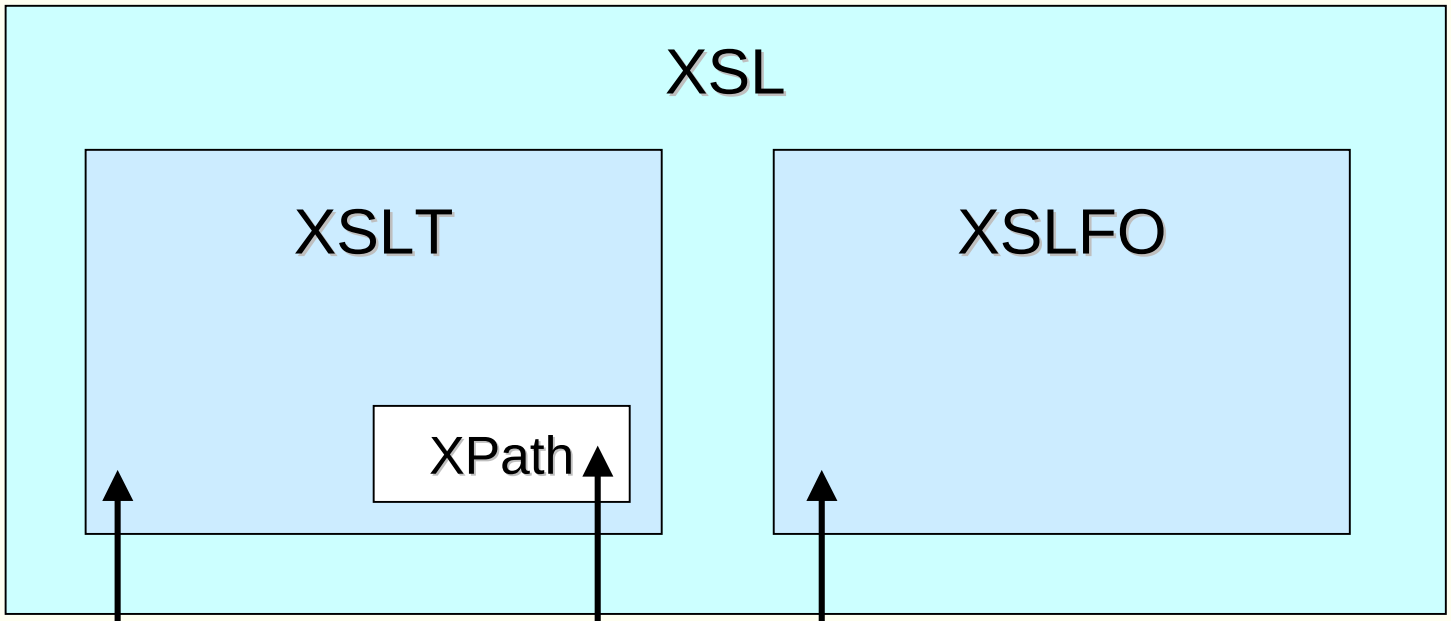
XSL stylesheet

- different views
- different layouts
- dynamic contents

XSL: key concepts

- Two separate processes:
  - transformation: **XSLT** (XSL **T**ransformations) using XPath
  - formatting: **XSLFO** (XSL **F**ormatting **O**bjects)
- Transformation
  - how an XML source tree should be transformed
  - recursive and declarative programming
- Formatting
  - how the XML result tree should be presented
  - specified using **f**ormatting **o**bjects (FOs)
- Status:
  - XSLT: W3C Recommendation (November 16, 1999)
    - XSLT 1.1: W3C Working Draft (December 12, 2000)
    - XSLT 2.0 Requirements: W3C Working Draft (February 14, 2001)
  - XSLFO: W3C Candidate Recommendation (November 21, 2000)

XSL = XSLT (with XPath) + XSLFO



locating language

transformation language
how a tree should be transformed

formatting vocabulary
how a (transformed) tree should be shown

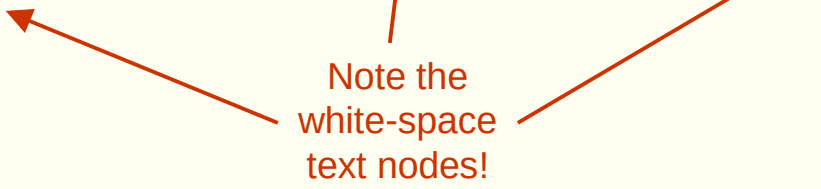
The XSLT data model

- An XSLT processor builds from the XML source file a tree structure consisting of 7 types of nodes:
  - element node
  - attribute node
  - text node
  - comment node
  - processing instruction node
  - root node
  - (namespace node)

- Beware: in a non-empty element
  - the text of that element is a child node of the element node
  - an attribute of that element is not a child node of the element node

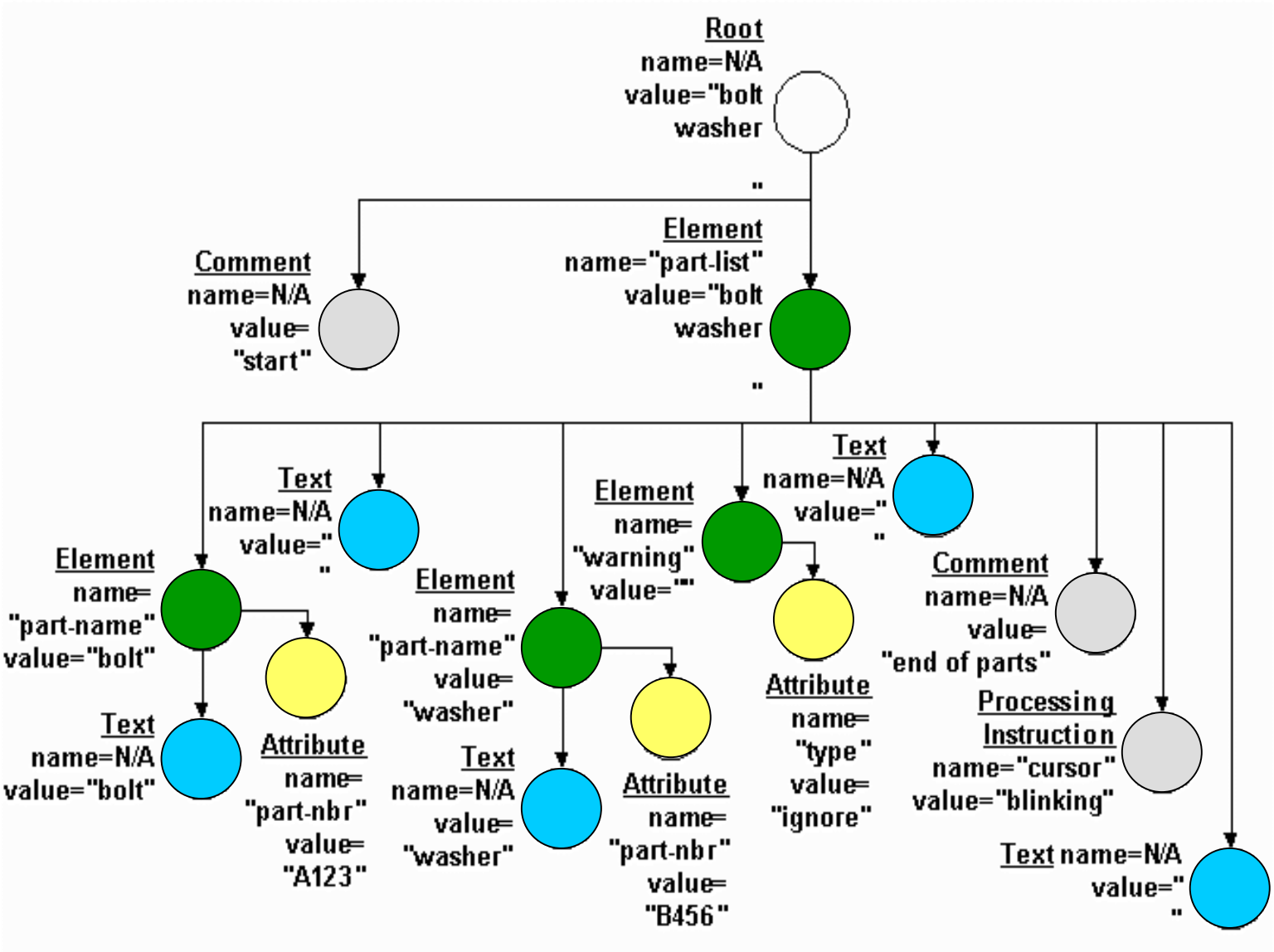
The XSLT data model

```
<?xml version="1.0"?>
<!--start-->
<part-list><part-name part-nbr="A123">bolt</part-name>Ã
<part-name part-nbr="B456">washer</part-name><warning type="ignore"/>Ã
<!--end of parts--><?cursor blinking?>Ã
</part-list>
```



Node type	Node name (example)	Node Value (example)
Element	part-name	bolt
Attribute	part-nbr	A123
Text	N/A	bolt
Comment	N/A	end of parts
Processing instruction	cursor	blinking
Root	N/A	boltÃ washerÃ Ã

The XSLT data model



The XSLT data model

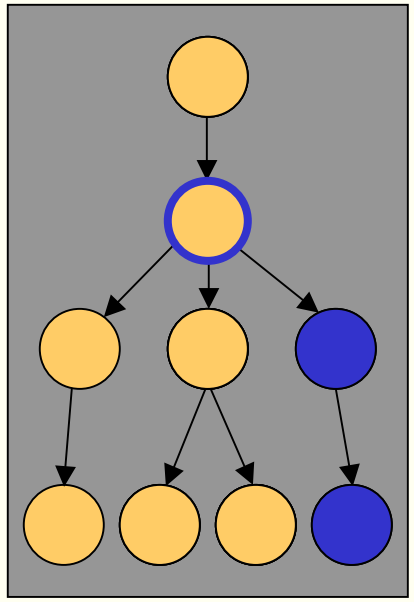
```
SHOWTREE Stylesheet - http://www.CraneSoftwrights.com/resources/
Processor: SAXON 6.1 from Michael Kay
1  Comment: {start}
2  Element 'part-list':
2.1  Element 'part-name' (part-list):
2.1.A  Attribute 'part-nbr': {A123}
2.1.1  Text (part-list,part-name): {bolt}
2.2  Text (part-list): {
}
2.3  Element 'part-name' (part-list):
2.3.A  Attribute 'part-nbr': {B456}
2.3.1  Text (part-list,part-name): {washer}
2.4  Element 'warning' (part-list):
2.4.A  Attribute 'type': {ignore}
2.5  Text (part-list): {
}
2.6  Comment (part-list): {end of parts}
2.7  Proc. Inst. 'cursor' (part-list): {blinking}
2.8  Text (part-list): {
}
```

XPath: XML Path Language

- Language for locating things in an XML document
  - finding elements, attributes, text, processing instructions, …
    by stepping through the document using a **location path**
  - location path is string-based rather than tag-based
    in order to be used in attributes and URLs
    - e.g. `<link ref="location path">`
  - also used:
    - in XPointer (part of XLL) to specify symbolic link addresses
    - in XQL to specify which things to retrieve from an XML document
- Location path
  - is built from **location steps** concatenated with /
    - `location step/location step/location step`
  - location step specifies a point in the document
    relative to another point in the document (the **context node**)

Location path

- Three possible starting positions
  - absolute
    - starting from the root "/"
  - relative from the context node
    - starting from the current node
  - from a well-defined position
    - using id() or document() functions



- Syntax of a location step:

  ***axis*::*node-test*[*predicate*]**

  - axis: in what direction to search from the context node
  - node-test: which set of nodes to consider along the axis
  - predicate: boolean expression that tests each node in that set

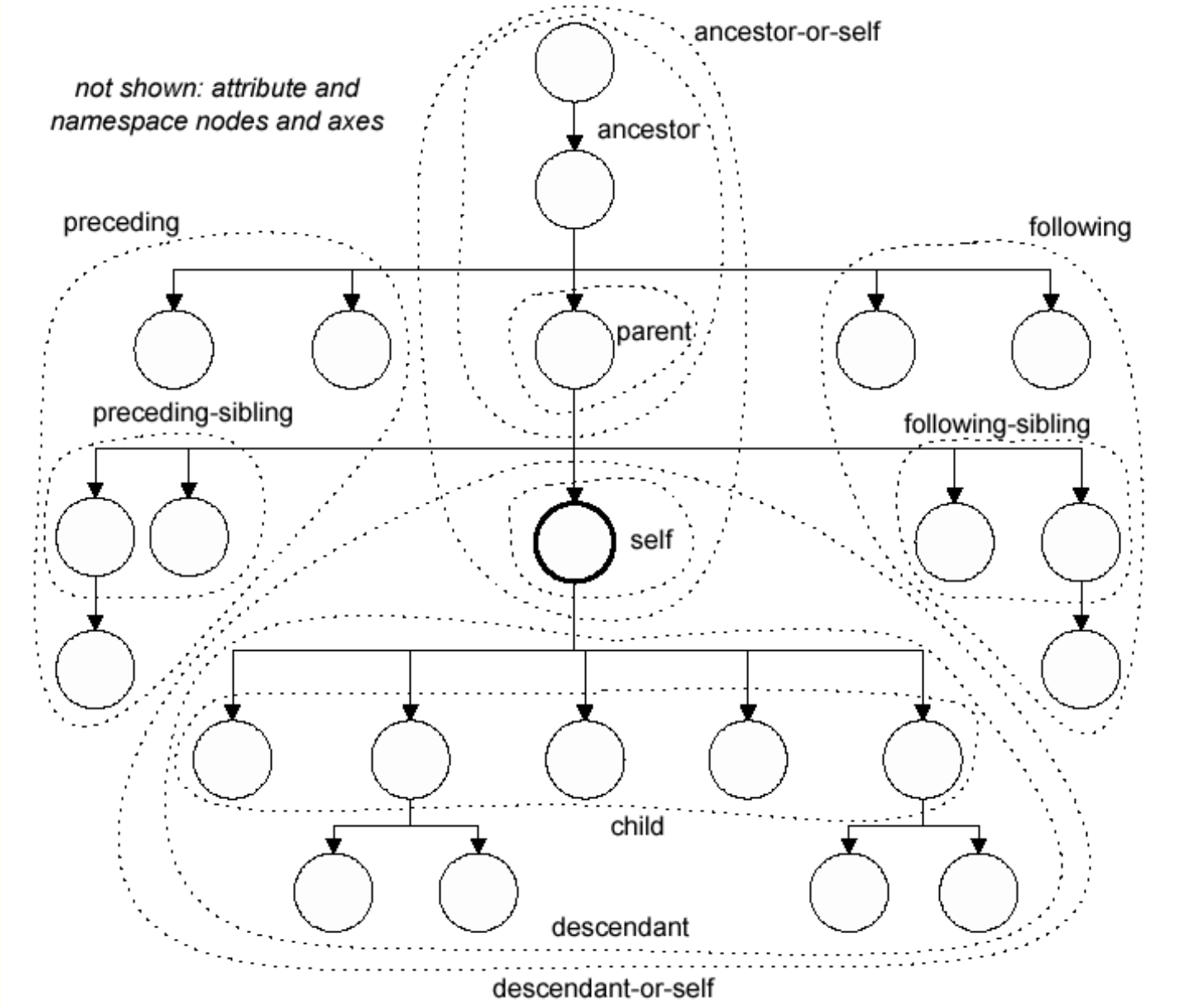
XPath: syntax

- Axis
  - **self**
  - **attribute**
  - **parent, child**
  - **ancestor, ancestor-or-self, descendant, descendant-or-self**
  - **preceding, preceding-sibling, following, following-sibling**
- Node-test
  - ***element name, attribute name, ****
  - **node(), text()**
  - **comment(), processing instruction()**
- Predicate
  - whole range of filtering predicates
    - **position(), first(), last(), …**

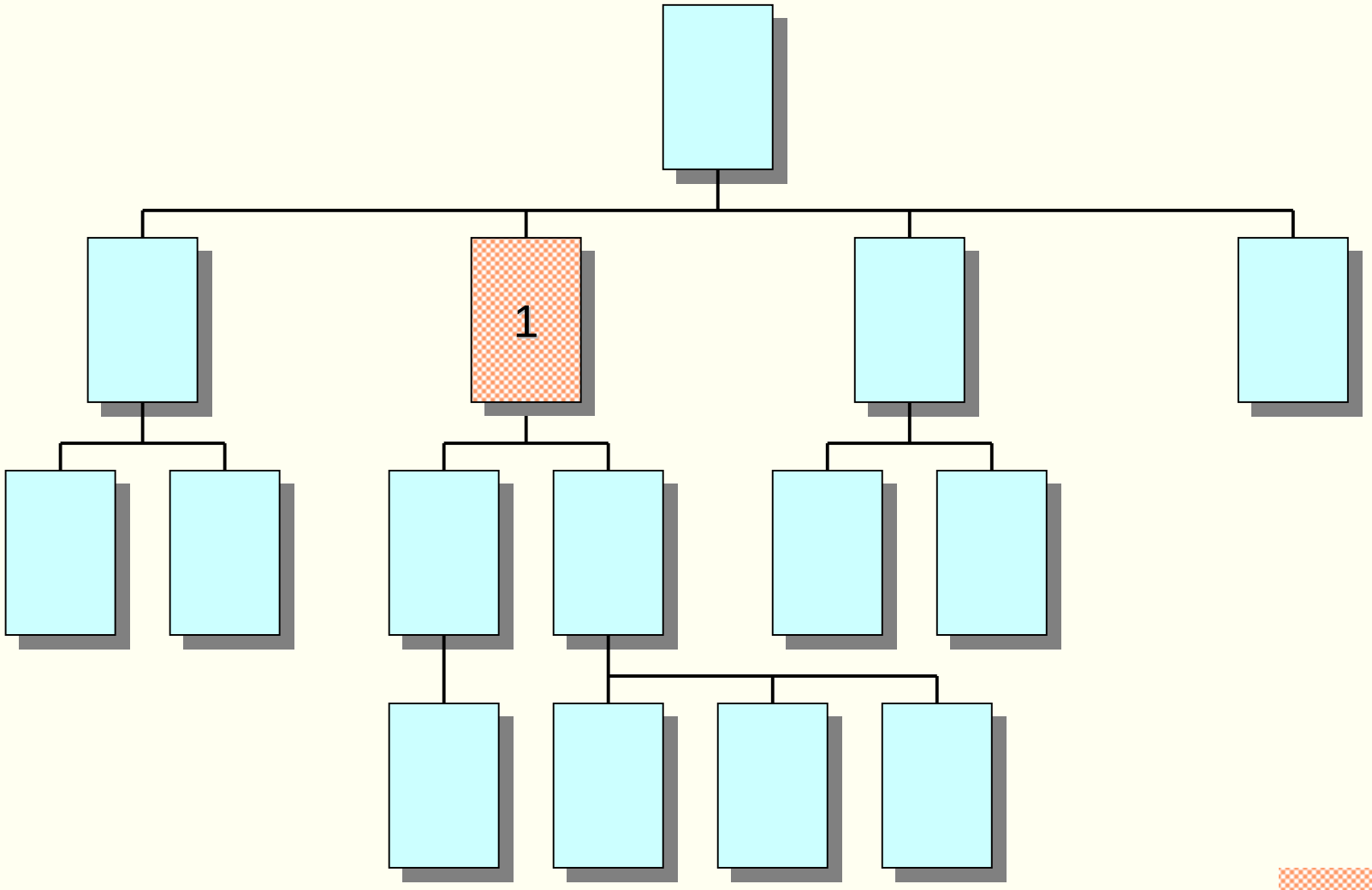
Axes

`self`	the context node
`child`	the immediate children of the context node
`descendant`	the children of the context node, the children of the children of the context node, the children of the children of the children of the context node, and so forth down to the leave nodes
`descendant-or-self`	the context node itself and its descendants
`parent`	the unique parent node of the context node
`ancestor`	the parent of the context node, the parent of the parent of the context node, the parent of the parent of the parent of the context node, and so forth up to the root node
`ancestor-or-self`	the ancestors of the context node and the context node itself
`following`	all nodes that start after the end of the context node, excluding attribute and namespace nodes
`following-sibling`	all nodes that start after the end of the context node and have the same parent as the context node
`preceding`	all nodes that start before the beginning of the context node, excluding attribute and namespace nodes
`preceding-sibling`	all nodes that start before the beginning of the context node and have the same parent as the context node
`attribute`	the attributes of the context node

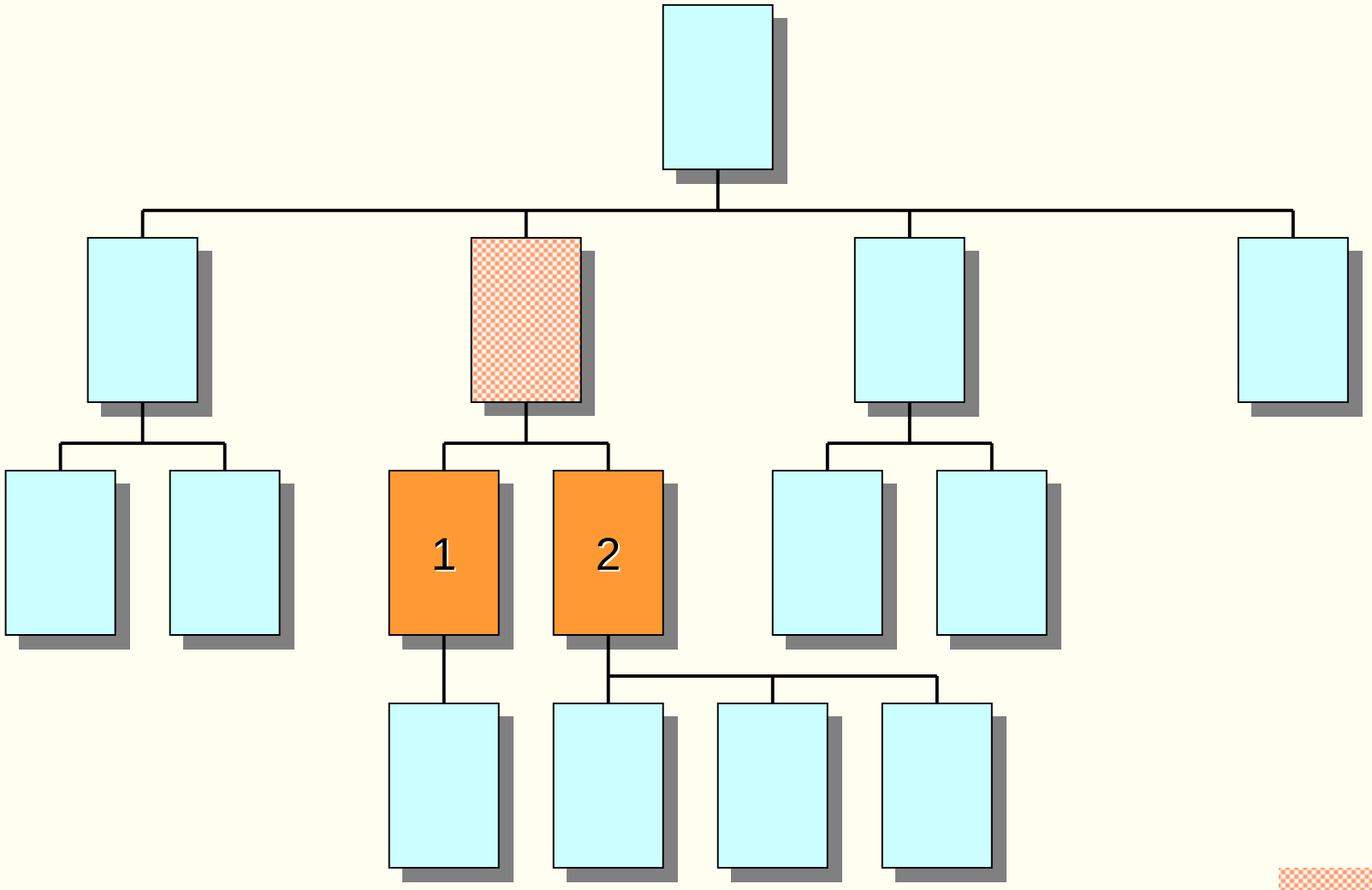
Axes


ancestor-or-self
not shown: attribute and namespace nodes and axes
ancestor
preceding
following
parent
preceding-sibling
following-sibling
self
child
descendant
descendant-or-self

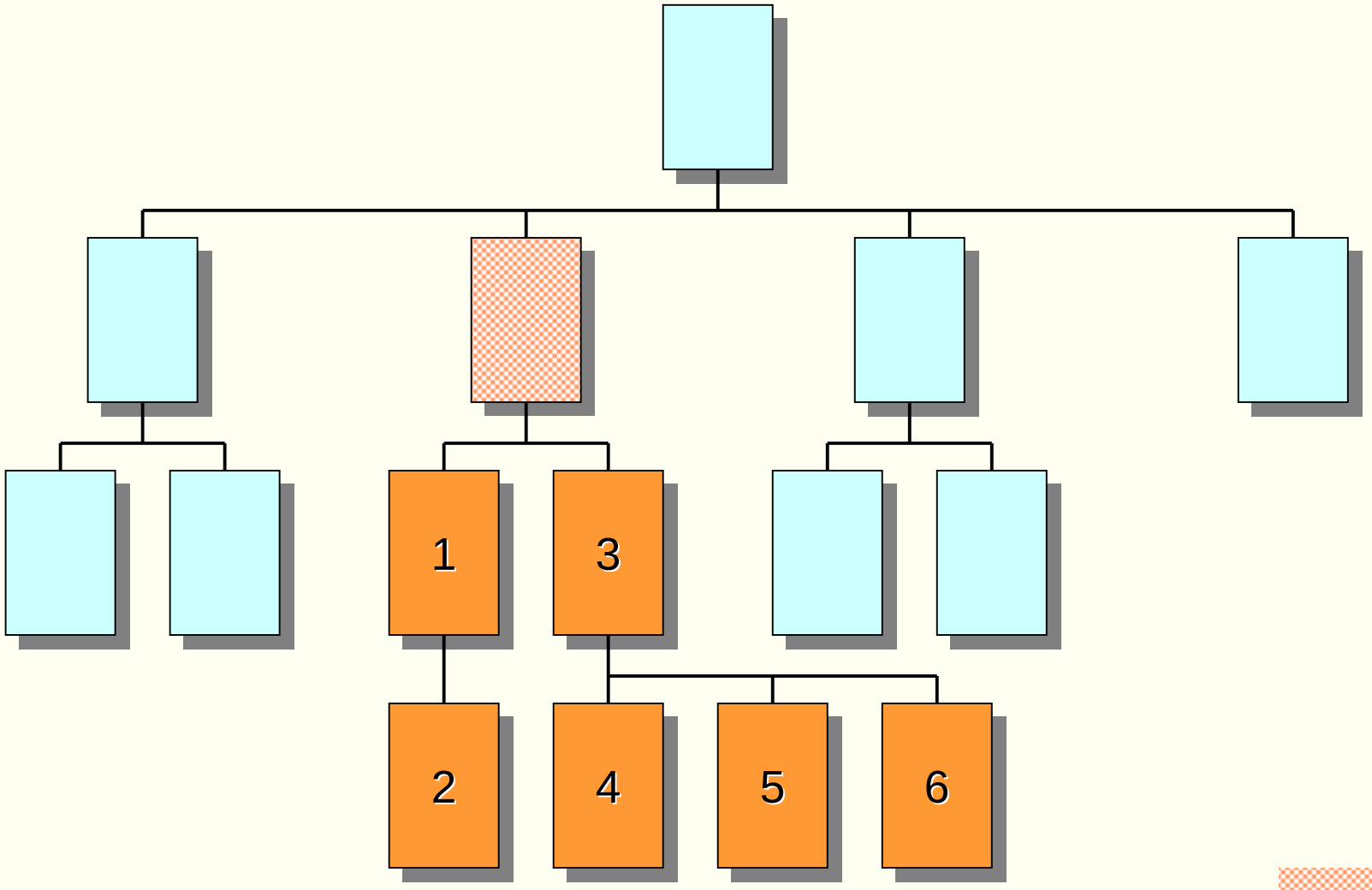
Axis `self`



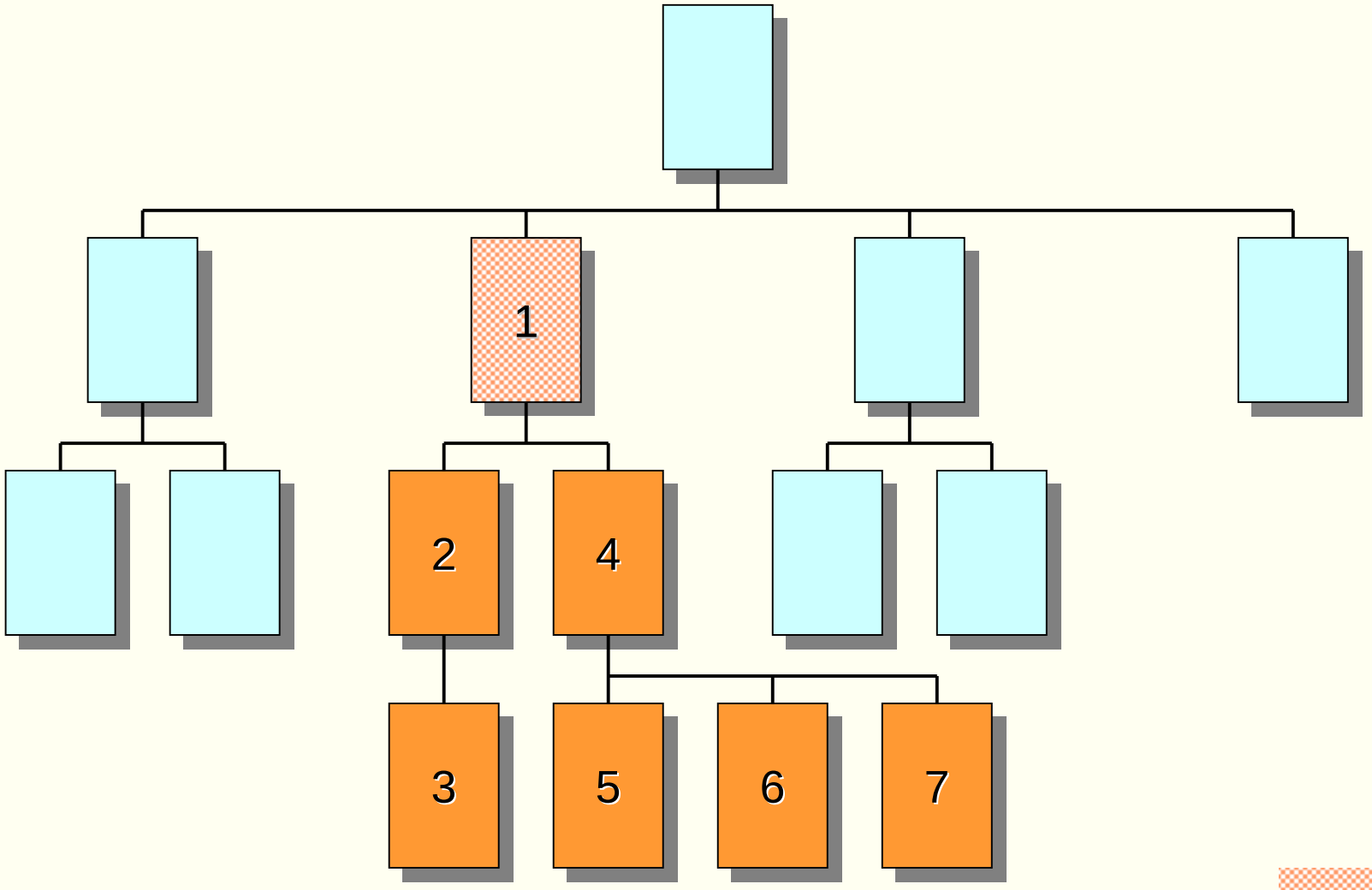
Axis `child`



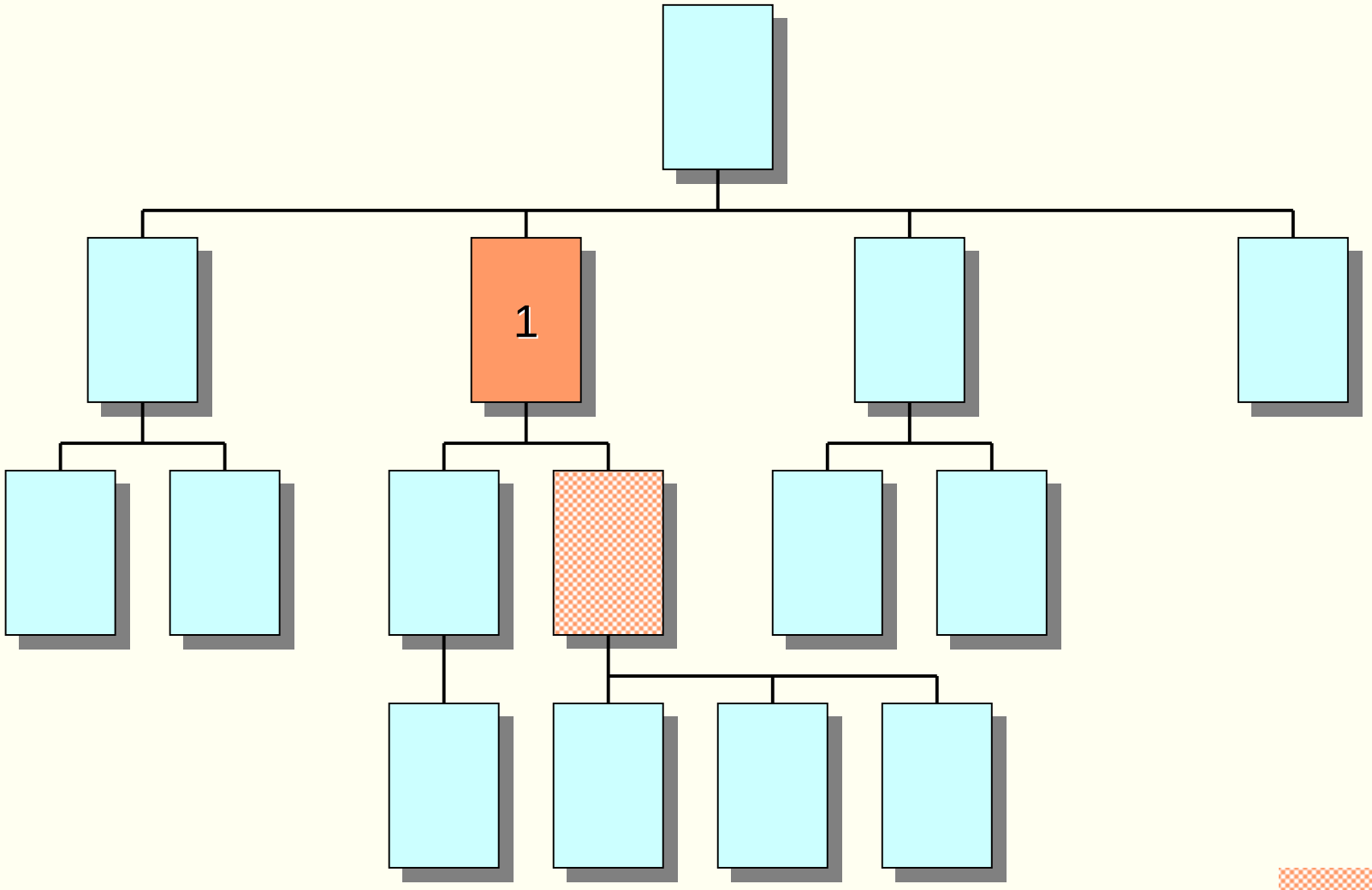
Axis **descendant**



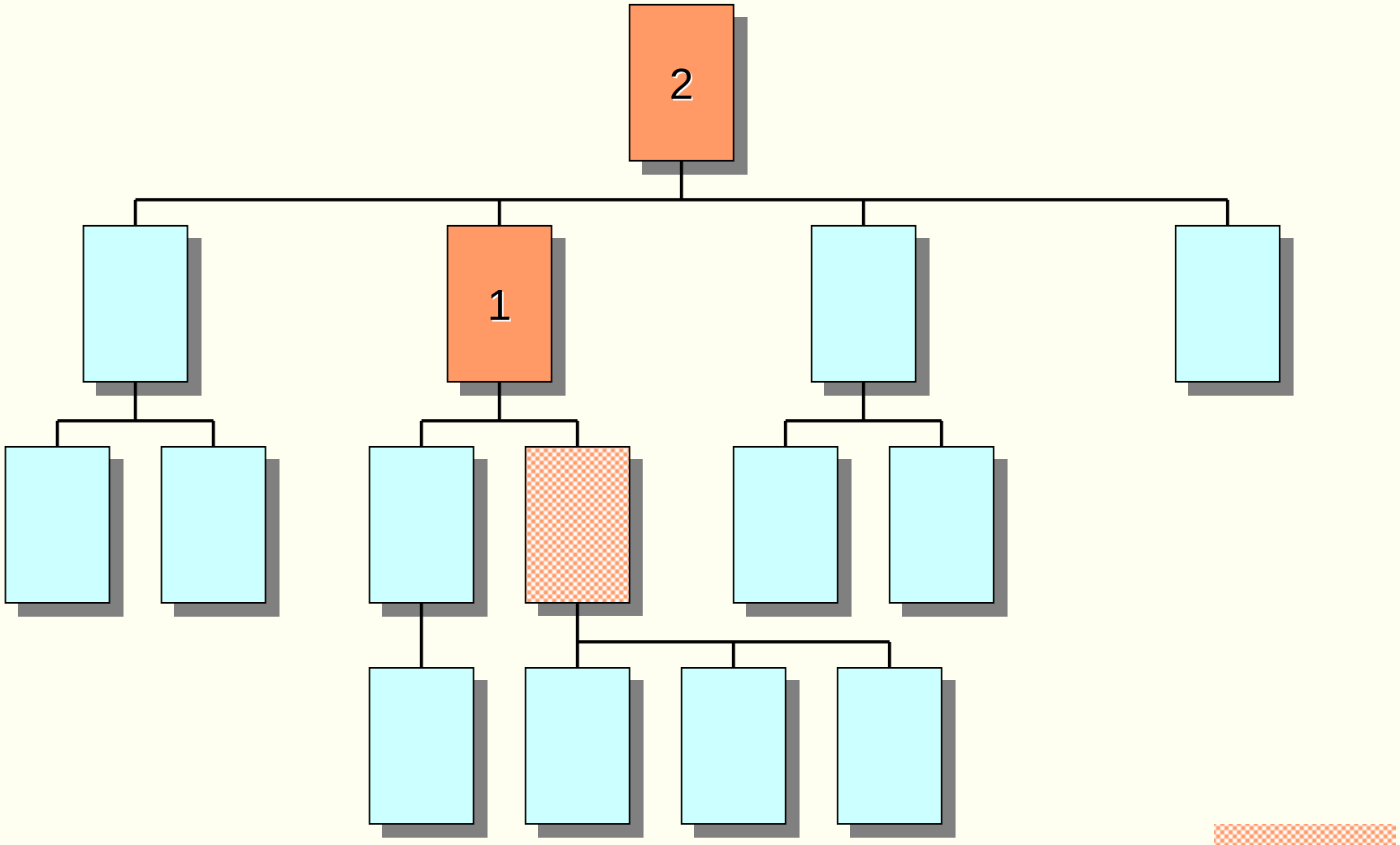
Axis `descendant-or-self`



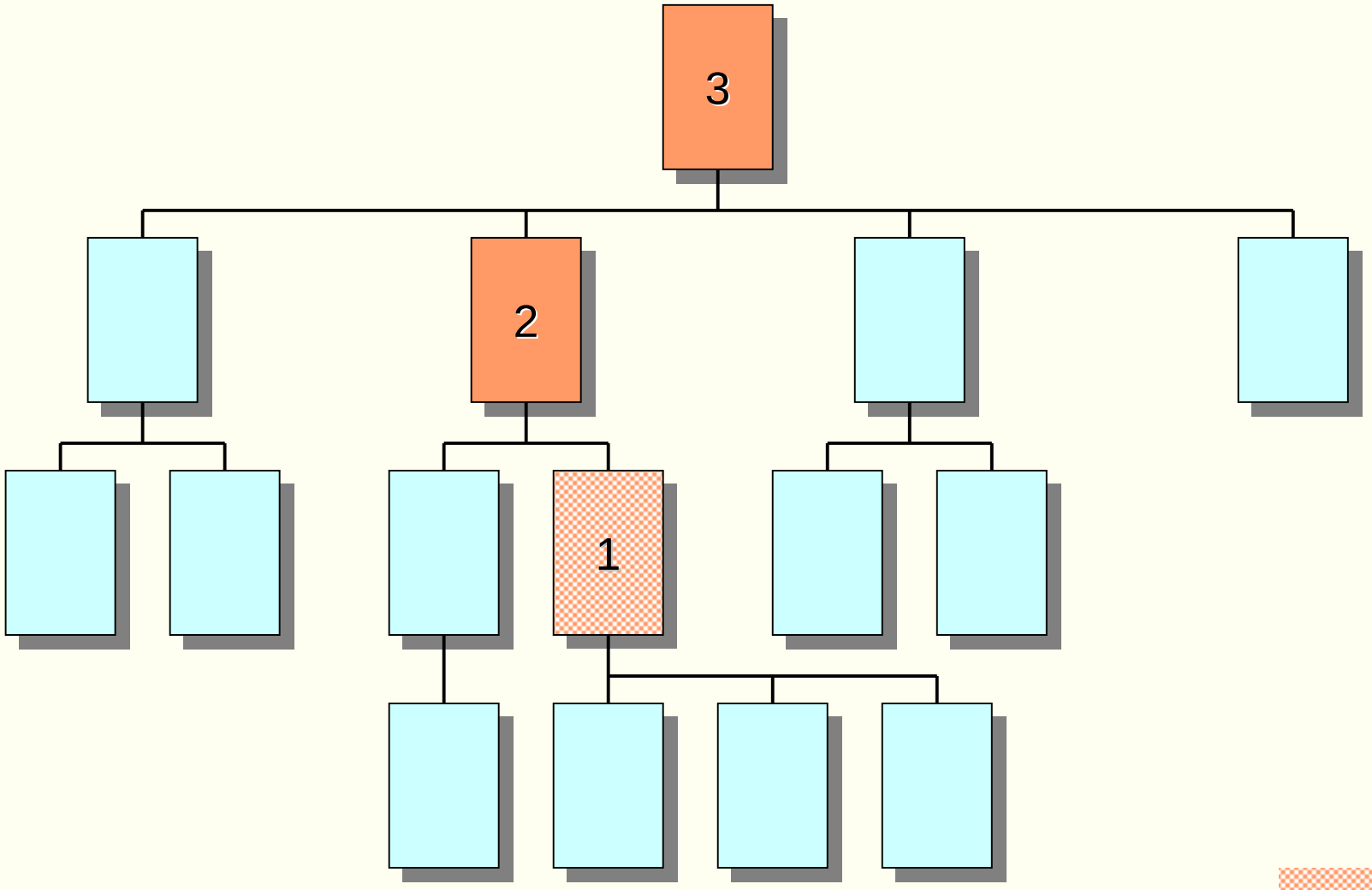
Axis **parent**



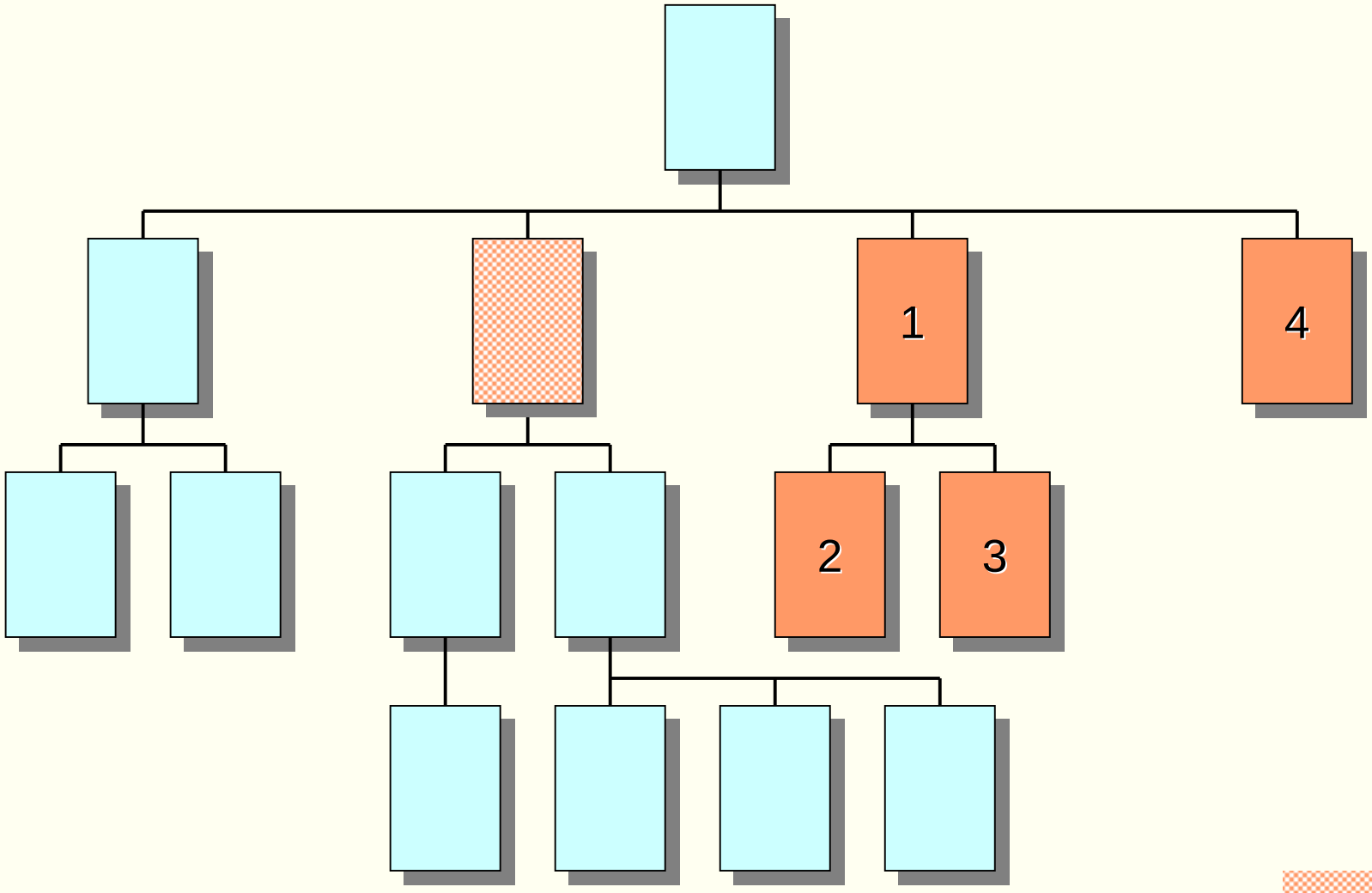
Axis **ancestor**



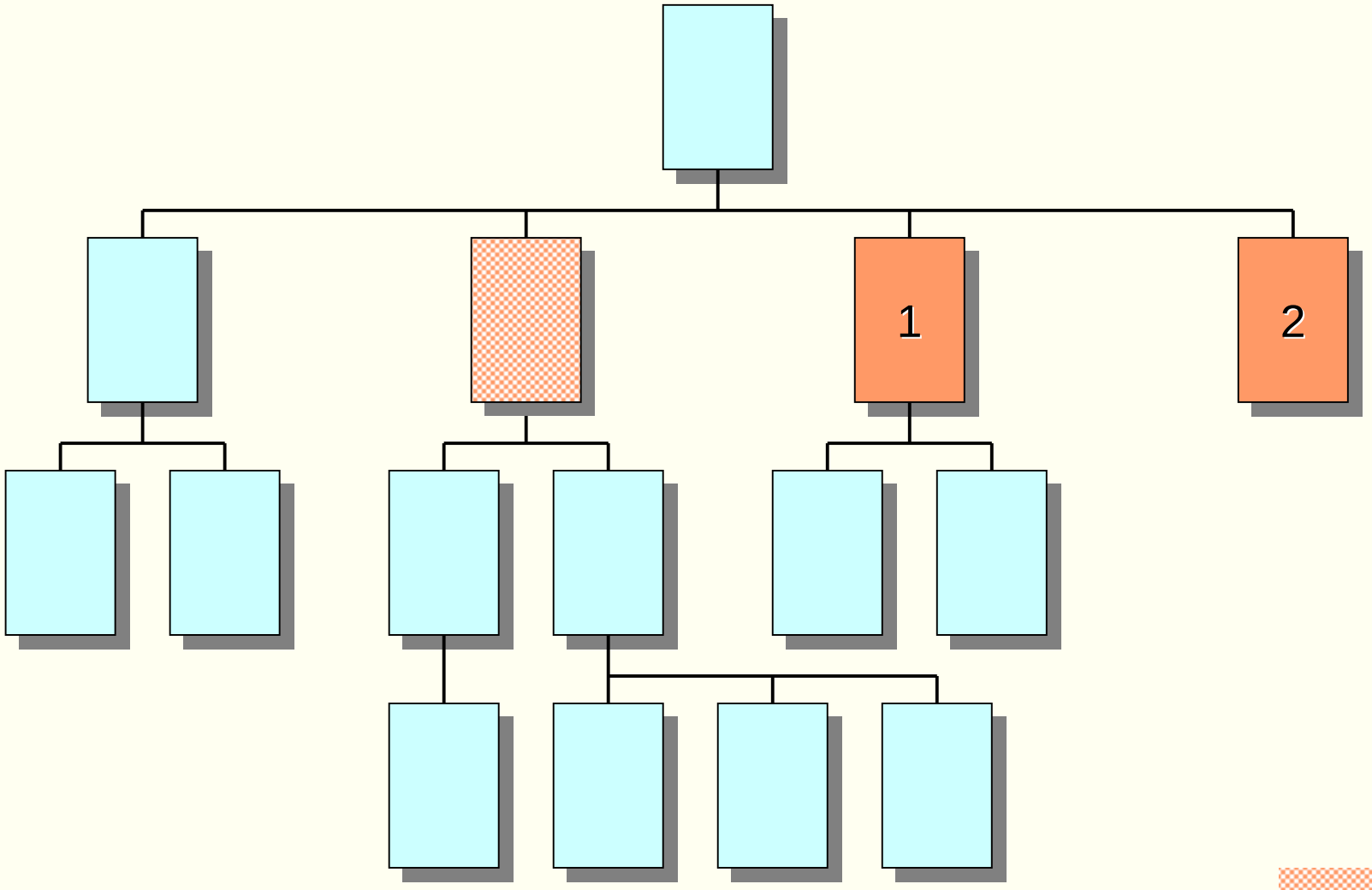
Axis `ancestor-or-self`



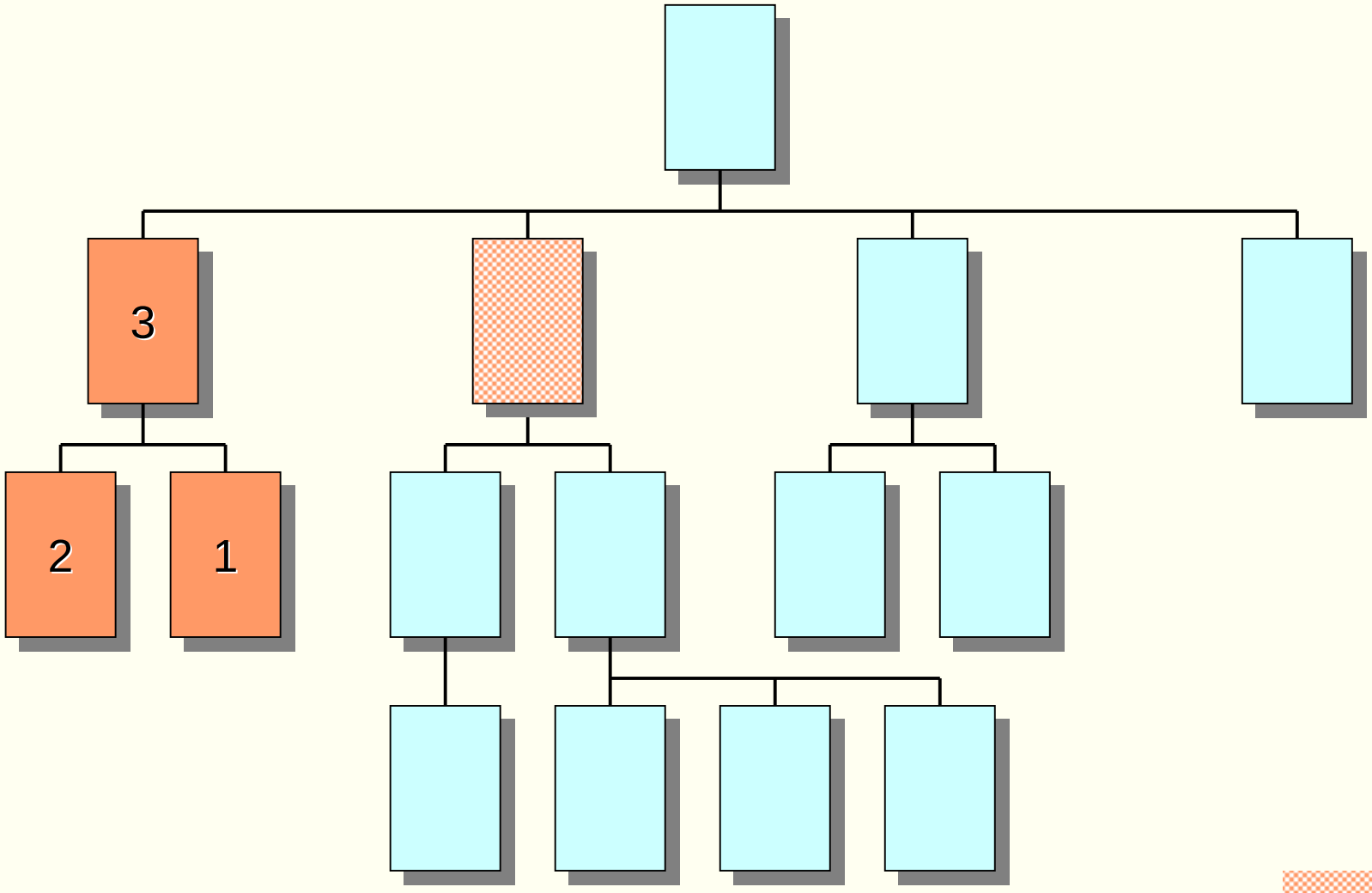
Axis `following`



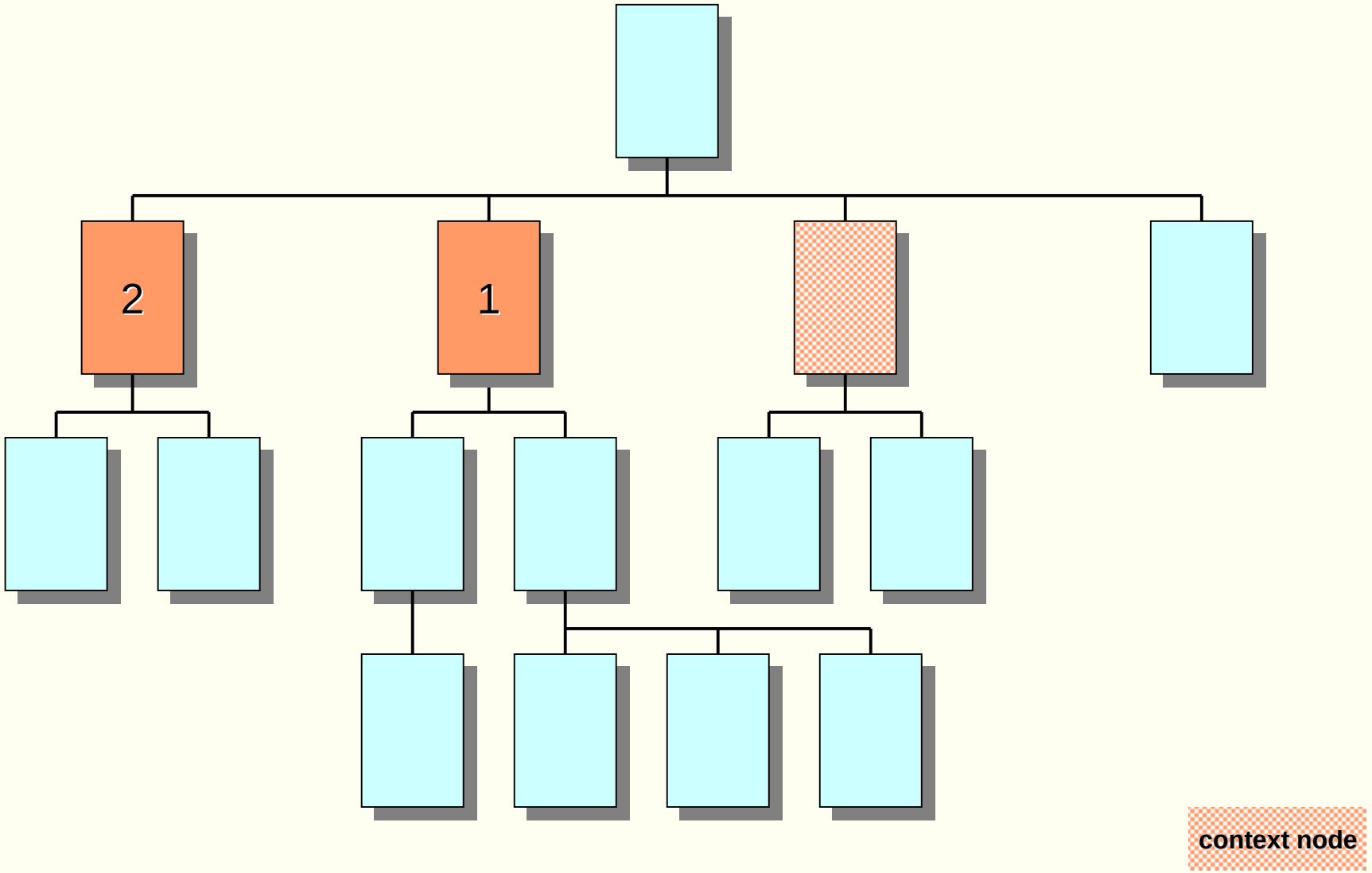
Axis `following-sibling`



Axis `preceding`



Axis `preceding-sibling`



Node-test

- on elements:
  - ***name***
  - *****
- on attributes:
  - ***name***
  - *****
- on node type:
  - **node()**
  - **text()**
  - **comment()**
  - **processing-instruction()**

Examples of node-tests

```
ancestor::*
ancestor-or-self::para
attribute::*
attribute::nr
child::*
descendant-or-self::node()
following-sibling::processing-instruction()
preceding::comment()
self::text()
```

Predicate

- boolean expression that tests each node in the set of nodes
- syntax:

  [*boolean-expression*]

  - **position(), first(), last(), …**
  - function on a node set that results in a boolean value
- Examples of the use of predicates

```
ancestor::para[position()=1]
ancestor::para[position()=last()]
ancestor::para[1]
child::para[attribute::type]
child::para[attribute::type="warning"]
```

Shorthand notations

`root()`	`/`
`child::`*`para`*	*`para`*
`child::text()`	`text()`
`attribute::`*`nr`*	`@`*`nr`*
`self::node()`	`.`
`parent::node()`	`..`
`/descendant-or-self::node()/`	`//`

Examples

- **`id("c725")`** the element with unique identifier c725
- **`root()`** *abbreviated to* **`/`** the root of the document
- **`child::para`** *abbreviated to* **`para`**

  `self::node()/para` *abbreviated to* **`./para`**
  - all of the para children of the context node
- **`parent::node()/para`** *abbreviated to* **`../para`**
  - all of the para children of the parent of the context node
- **`/descendant::node()/para[position()=5]`**

  abbreviated to **`//para[5]`**
  - the fifth para element anywhere in the document
- **`attribute::align`** *abbreviated to* **`@align`**
  - the align attribute of the context node

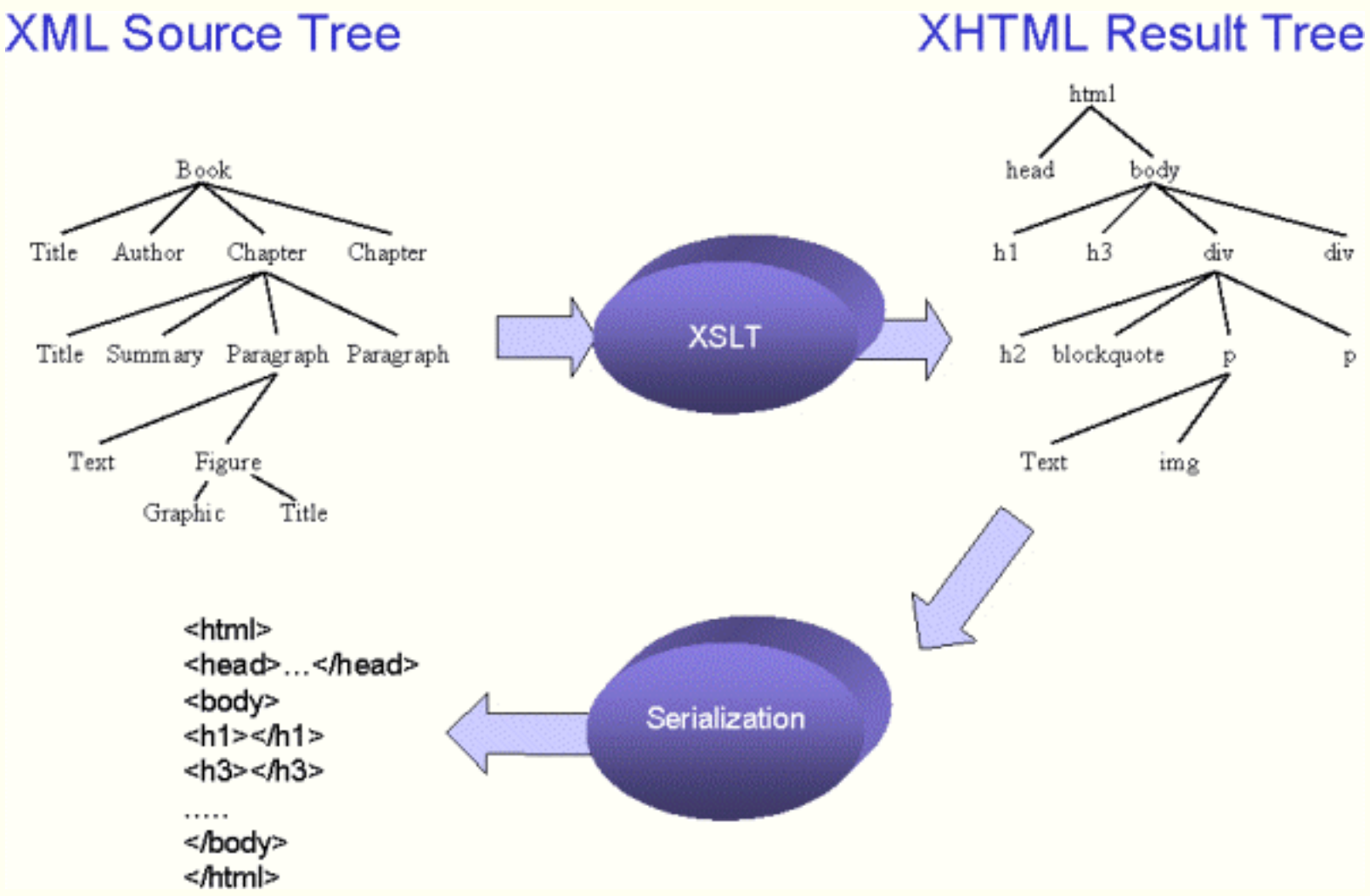
Examples

- **`list/item`**
  - all of the item children of all of the list children of the context node
- **`list//para`**
  - all of the para elements (nested arbitrarily deep) under all of the list children of the context node. In other words, this location path matches list/item/para, list/item/note/para, …
- **`note[@type="warning"]`**
  - only note children that have a type attribute with the value "warning"
- **`section[title="Introduction"]`**
  - all the section children with a title of "Introduction"
- **`//section[title="Examples"]/example[last()]`**
  - the last example element in all section elements (anywhere in the document) with a title element of "Examples"

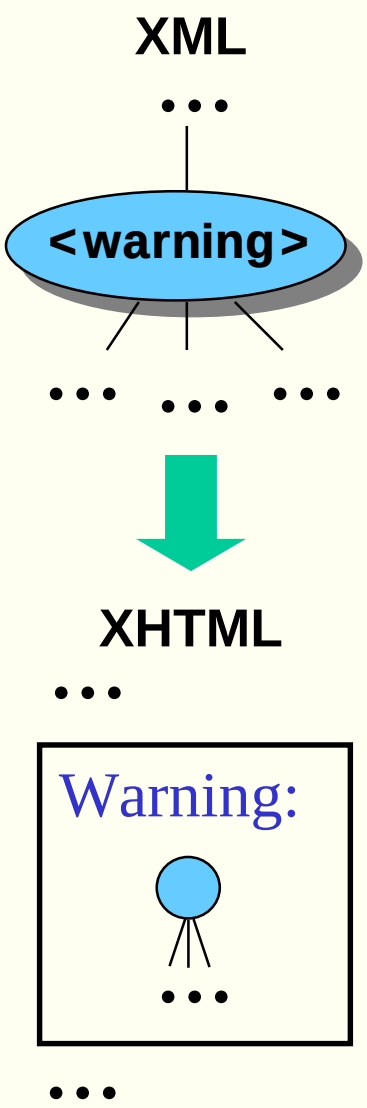
Exercise

```
para
./para
../para
.//para
//para
self()::para
@type
../@type
../para/@type
../para[@type]
para[@type="warning”]/@align
para[@type][3]
id(@ref)/@nr
```

XSLT = transformation + serialization



Processing model of XSLT



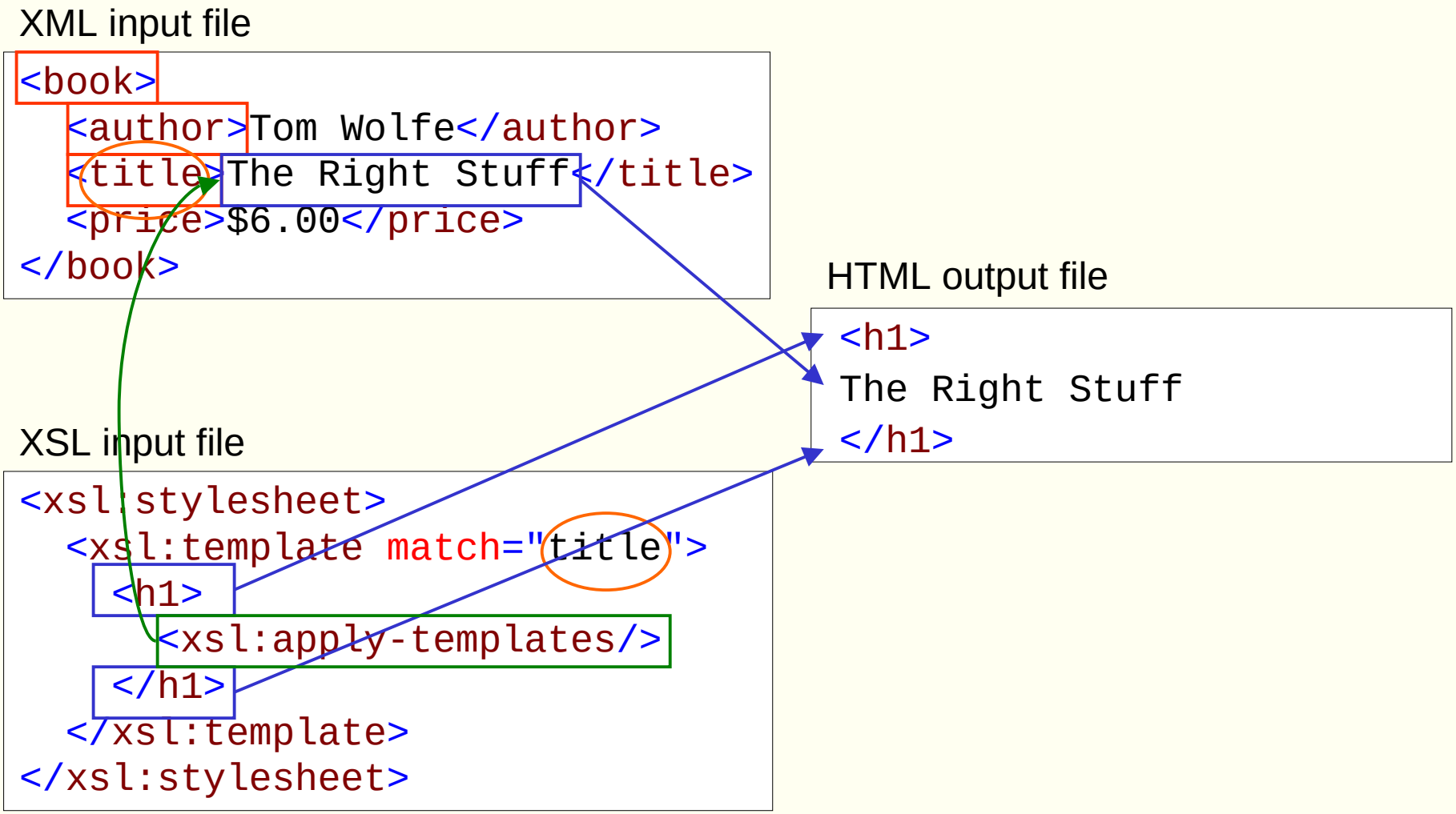

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0">

  <!-- pattern in source tree    -->
  <xsl:template match="warning">
  <!-- element(s) in result tree -->
  <p style="font-size=24pt;
            font-family=serif">
    Warning:
    <xsl:apply-templates/>
  </p>
  </xsl:template>

</xsl:stylesheet>
```

rule pattern

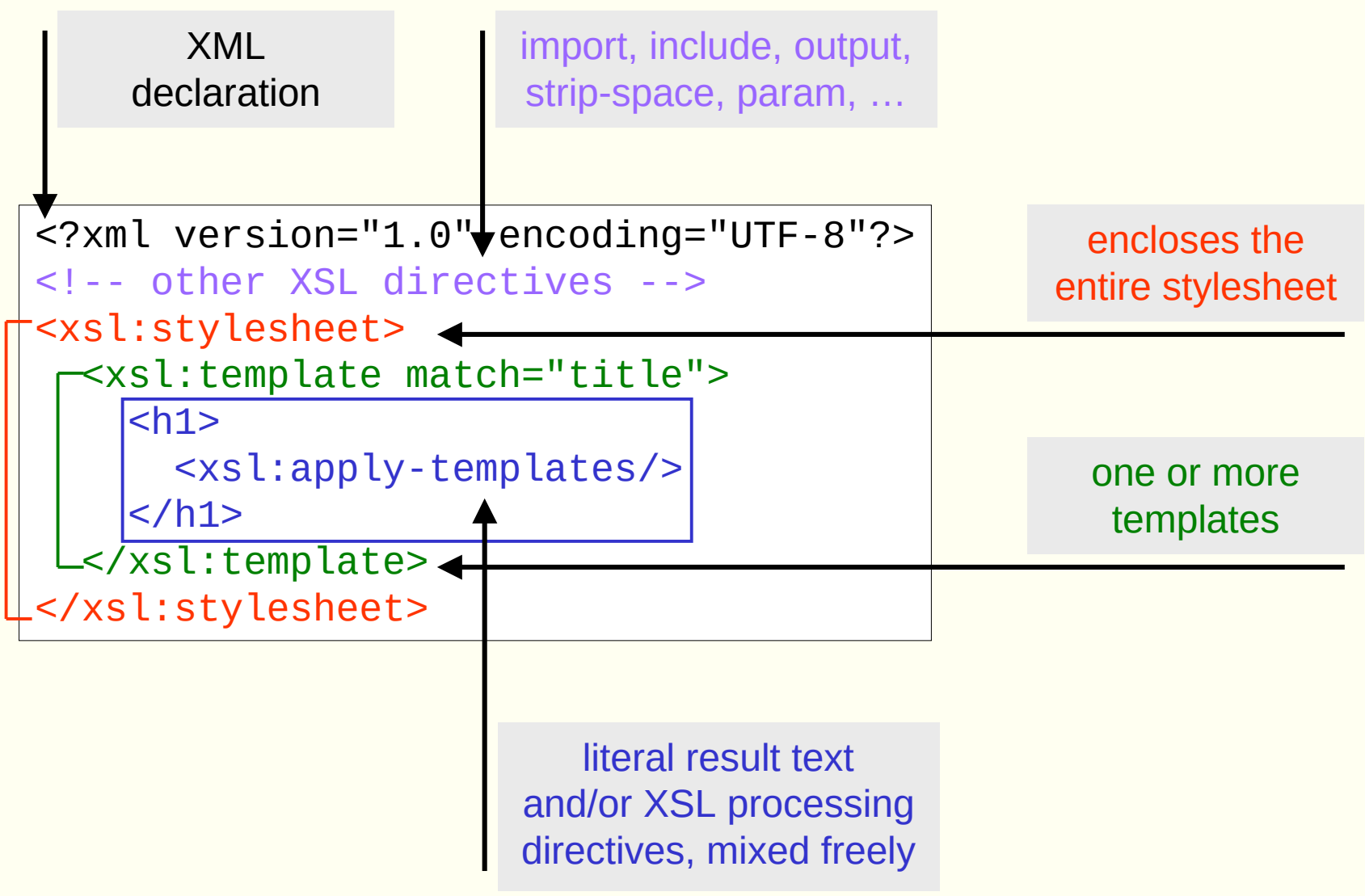
rule action

Stylesheet example: generating HTML

- Insert only the title as a heading 1 in the HTML file



Anatomy of an XSLT stylesheet



xsl:template element

```
<xsl:template match="match-expression">
  <!-- literal result text, XSLT elements -->
</xsl:template>
```

- Specifies:
  - a match-expression that defines when this rule will be used (this is an XPath expression)
  - literal result text to copy to output, and/or other XSLT elements to cause other actions (copying data from the source tree to the result tree)
  - priority and/or template-name (not covered)
- A stylesheet has one or more templates
  - when `<xsl:template>` is missing, and literal result text is found in the stylesheet, a `match="/"` template is assumed to be present

Let's see if the stylesheet works!

What's missing from the example?

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

- Specifies:
  - a string that specifies a particular vocabulary
    (in this case, the standard W3C XSLT 1.0 vocabulary)
    - pre-standard MS XSL: `http://www.w3.org/TR/WD-xsl`
    - XSL processors may vary in behaviour depending on the string you use
  - a namespace prefix used to label every XSLT statement
  - a version number attribute
    - present XSL standard: version 1.0, future XSL standard: version 1.1
- There must be exactly one `<xsl:stylesheet>` element that encloses all other stylesheet elements

Let's see if the stylesheet works!

XSL input file (corrected)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="title">
    <h1>
      <xsl:apply-templates/>
    </h1>
  </xsl:template>
</xsl:stylesheet>
```

HTML output file

```
<?xml version="1.0" encoding="utf-8"?>
  Tom Wolfe
  <h1>The Right Stuff</h1>
  $6.00
```

why does it have the contents of
the `author` and the `price` elements?

xsl:apply-templates element

- Purpose: invokes rules as appropriate
  to process *children of the current node*
  - the template match rules you define
  - default rules built-in to the XSL processor
- In our stylesheet:

```
<xsl:template match="title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
```

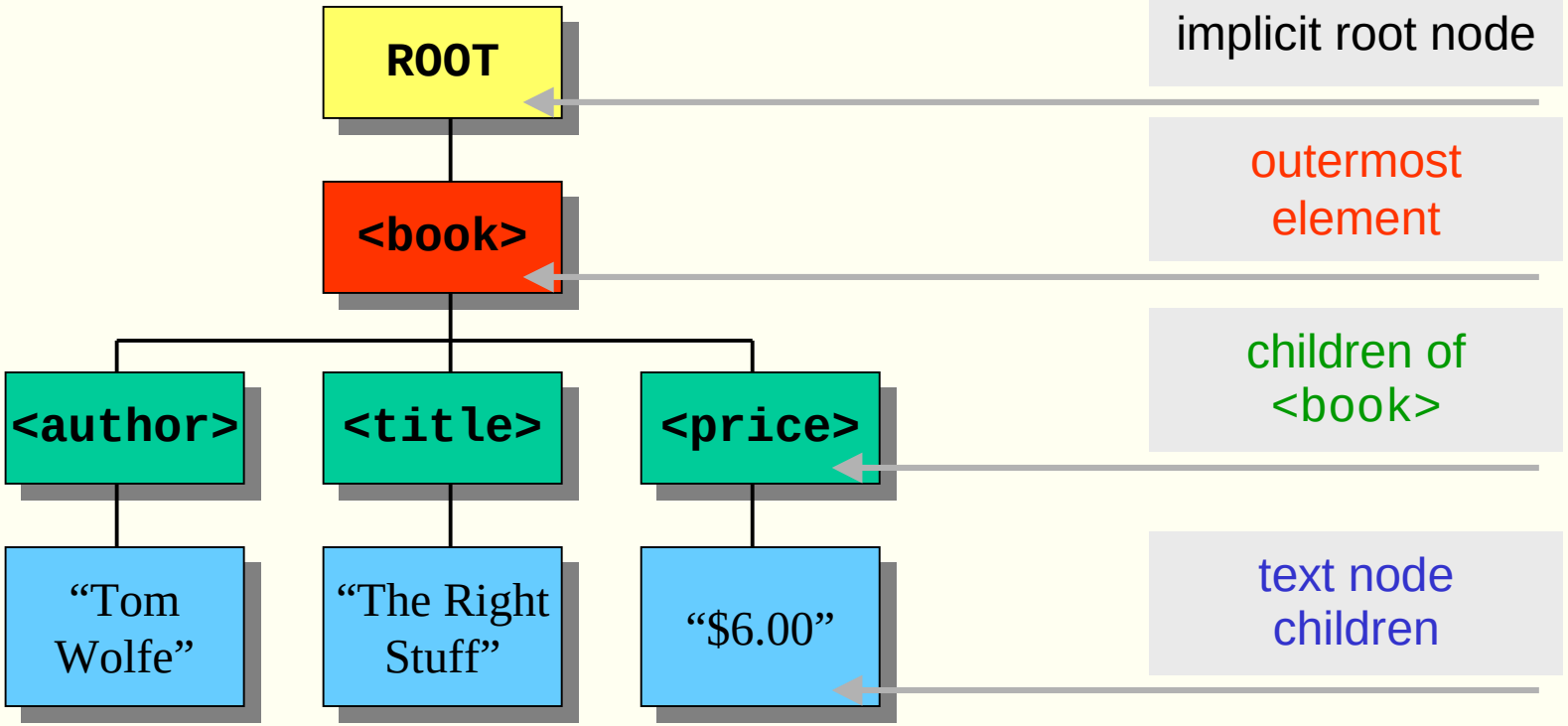
  used in the template defined for the `<title>` element

- *So what are the children of the current node here and how are which rules going to be invoked?*

How does the XSL processor see the XML?

```
<book>
  <author>Tom Wolfe</author>
  <title>The Right Stuff</title>
  <price>$6.00</price>
</book>
```

a tree representation is created by parsing the document

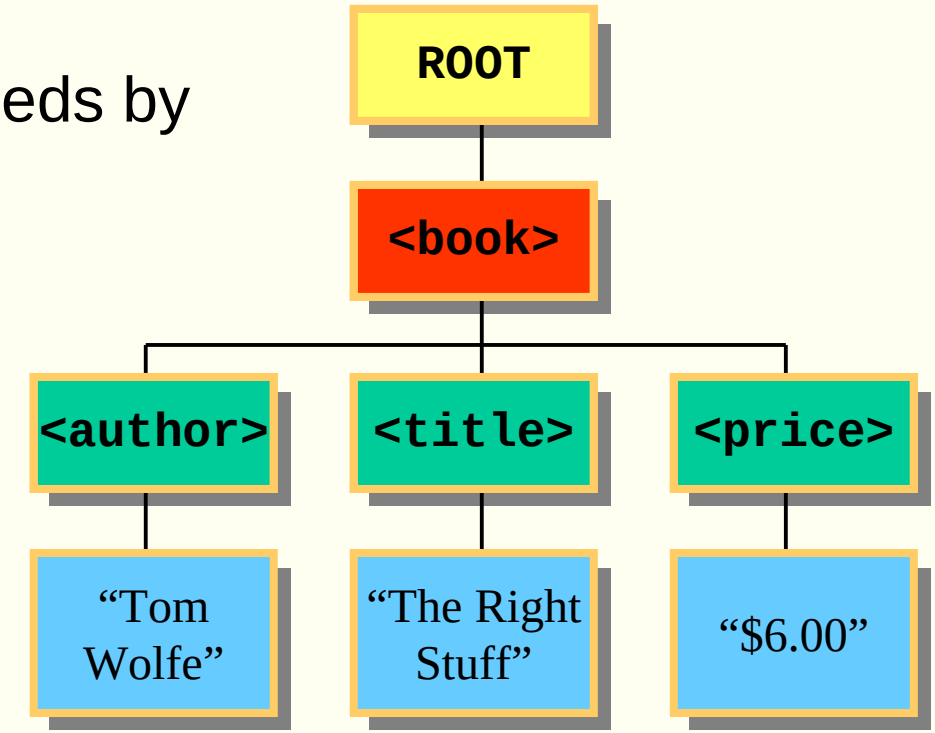


Built-in default template rules

- For element (and root) nodes:
  "apply-templates to every child"

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

- Template processing proceeds by "walking through the tree", in a recursive fashion



Built-in default template rules

- For text and attribute nodes:
  "copy the text to output"

```
<element>text<element>
<element attribute="text"/>
```

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

- For processing-instruction nodes and comment nodes:
  "do nothing"

```
<?xml version="1.0"?>
<!-- comment -->
```

```
<xsl:template match="processing-instruction()|comment()">
</xsl:template>
```

or also

```
<xsl:template match="processing-instruction()|comment()"/>
```

How does this apply to the example?

- Since there is no explicit template for the root, the built-in rule is invoked
- … which causes the template for `<book>` to be applied
- Since there is no explicit template for `<book>`, the built-in rule is invoked
- … which causes the templates for `<author>`, `<title>` and `<price>` to be applied
- Since there is no explicit template for `<author>`, the built-in rule is invoked
- ... which causes the built-in rule for the text child of `<author>` to be invoked
- ... which writes out the text of the element `<author>`. **OOPS!**
- Since there is an explicit template for `<title>`, this rule is executed
- … which writes out the HTML tag `<h1>`
- … followed by the invocation of the built-in rule for the text child of `<title>`
- ... which writes out the text of the element `<title>`
- … followed by writing out the HTML tag `</h1>`.
- Since there is no explicit template for `<price>`, the built-in rule is invoked
- ... which causes the built-in rule for the text child of `<price>` to be invoked
- ... which writes out the text of the element `<price>`. **OOPS!**

Overriding built-in default template rules

- How can we stop the built-in rule being invoked for the `<author>` and `<price>` text children?

  solution: override them with a template that does nothing

```
<xsl:template match="author">
</xsl:template>
<xsl:template match="price">
</xsl:template>
```

or

```
<xsl:template match="author|price">
</xsl:template>
```

or also

```
<xsl:template match="author|price"/>
```

Let's see if the stylesheet works!

XSL input file (corrected)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform“
  version="1.0">
  <xsl:template match="title">
    <h1>
      <xsl:apply-templates/>
    </h1>
  </xsl:template>
  <xsl:template match="author|price"/>
</xsl:stylesheet>
```

HTML output file

```
<?xml version="1.0" encoding="utf-8"?>

  <h1>The Right Stuff</h1>
```

What if we have to delete a lot of elements?

```
<xsl:template match="title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
<xsl:template match="author|price"/>
```

instead of deleting
the elements we do not
want to have in the result tree

```
<xsl:template match="book">
  <h1>
    <xsl:apply-templates select="title"/>
  </h1>
</xsl:template>
```

we can apply
template rules selectively
(i.e. just for the title element)

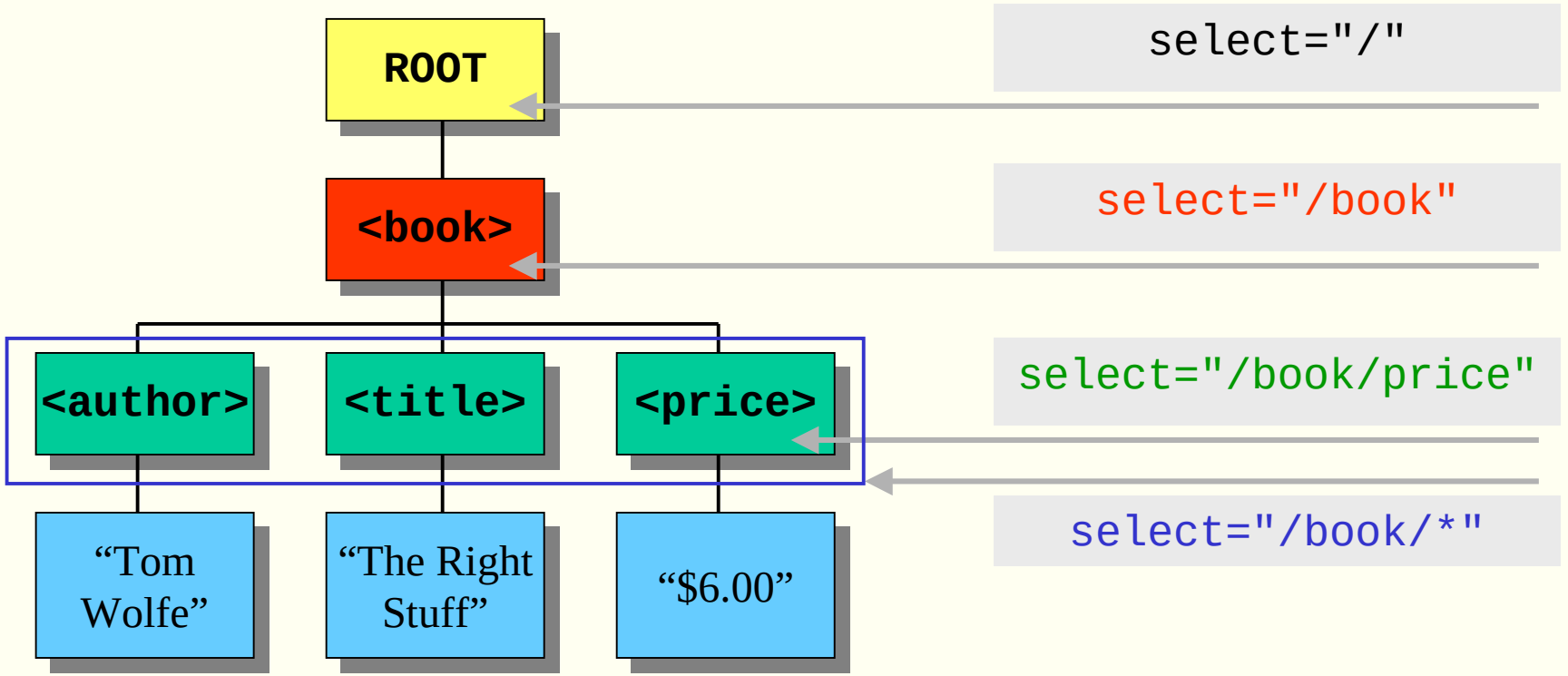
```
<xsl:template match="book">
  <h1>
    <xsl:value-of select="title"/>
  </h1>
</xsl:template>
```

or we can pull out
the text data from the child
element we do want to include

How do we select something in the tree?

```
<book>
  <author>Tom Wolfe</author>
  <title>The Right Stuff</title>
  <price>$6.00</price>
</book>
```

a tree representation is created by parsing the document



e.g. `<xsl:value-of select="/book/price"/>` returns "$6.00"

Let's see if the stylesheet works!

XSL input file (alternative)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:template match="book">
    <h1>
      <xsl:apply-templates select="title"/>
    </h1>
  </xsl:template>
</xsl:stylesheet>
```

HTML output file

```
<?xml version="1.0" encoding="utf-8"?><h1>The Right Stuff</h1>
```

- How to get rid of the XML declaration?
  - by using the `output` XSL directive

Let's see if the stylesheet works!

XSL input file (improved)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="book">
     <h1>
       <xsl:apply-templates select="title"/>
     </h1>
  </xsl:template>
</xsl:stylesheet>
```

HTML output file

```
<h1>The Right Stuff</h1>
```

xsl:output element

- Purpose: directive to control the serialization of the result tree
- In the example stylesheet:

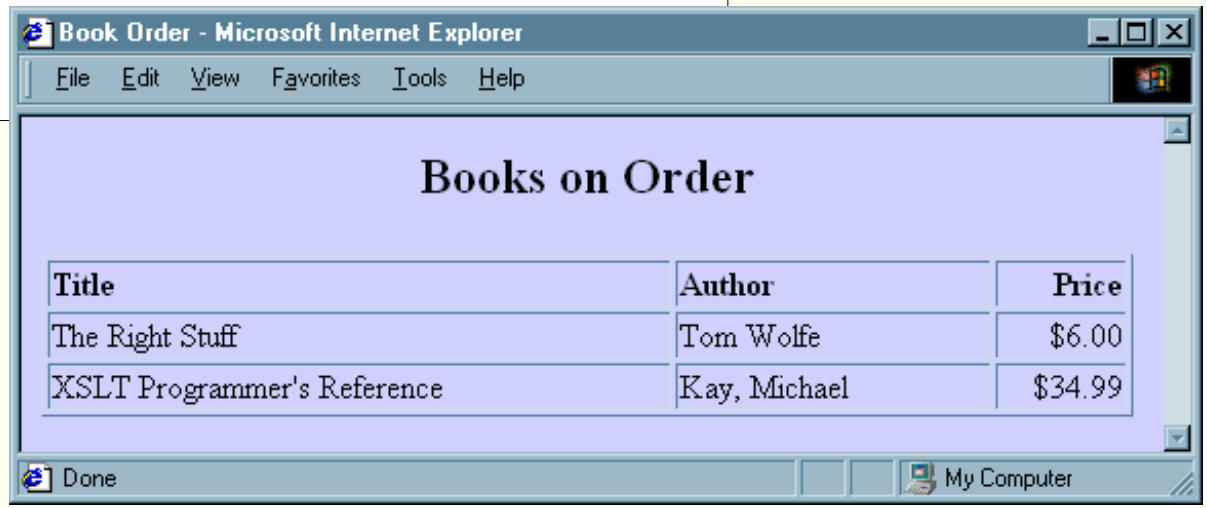
```
<xsl:output method="html"/>
```

  - methods:
    - `XML` output as XML
    - `HTML` output as HTML
    - `text` output as text
- Other parameters:

```
<xsl:output indent="yes"/>
<xsl:output version="1.0"/>
<xsl:output standalone="yes"/>
<xsl:output encoding="ISO-8859-1"/>
<xsl:output omit-xml-declaration="no"/>
```

Stylesheet example: generating HTML page

```
<?xml version="1.0" encoding="UTF-8"?>
<book-order>
  <book>
    <author>Wolfe, Tom</author>
    <title>The Right Stuff</title>
    <price>$7.99</price>
  </book>
  <book>
    <author>Kay, Michael</author>
    <title>XSLT Programmer's Reference</title>
    <price>$34.99</price>
  </book>
</book-order>
```



What do we need to do?

Œ Generate prologue of the result HTML file

- start the `<html>` element
- create `<head>` and `<title>` elements
- start the `<body>` element
- start the `<table>` element, with captions and column headings

all this must be done *first*, and just *once*

- Generate a row of data for each `<book>`
  - for any number of books in the source XML file

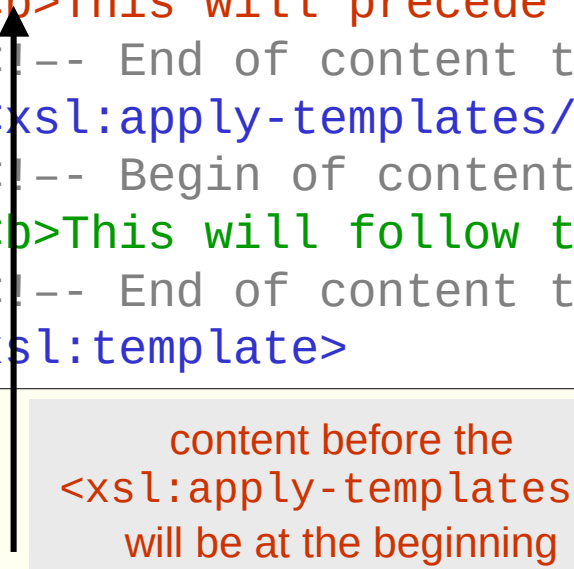
Ž Generate epilogue of the result HTML file

- close the `<table>` element
- close the `<body>` and `<html>` elements

this must be done *last*, and just *once*

One-time content at beginning and end

```
<xsl:output method="html"/>
<xsl:template match="/">
  <!-- Begin of content that precedes the data -->
  <b>This will precede the data.</b>
  <!-- End of content that precedes the data   -->
  <xsl:apply-templates/>
  <!-- Begin of content that follows the data  -->
  <b>This will follow the data.</b>
  <!-- End of content that follows the data    -->
</xsl:template>
```



content before the `<xsl:apply-templates/>` will be at the beginning

content after the `<xsl:apply-templates/>` will be at the end

- Technique: template for the root
  - there is only one root element
  - it's always the first node to be processed

  à use for once-only content at beginning or end

One-time content at beginning and end

```
<xsl:template match="/">
  <html>
    <head>
      <title>Book Order</title>
    </head>
    <body bgcolor="#d0d0ff">
      <table border="border">
        <caption><h2>Books on Order</h2></caption>
        <thead>
          <tr>
            <th align="left">Title</th>
            <th align="left">Author</th>
            <th align="right">Price</th>
          </tr>
        </thead>
        <tbody>
          <xsl:apply-templates/>
        </tbody>
      </table>
    </body>
  </html>
</xsl:template>
```

PROLOGUE

EPILOGUE

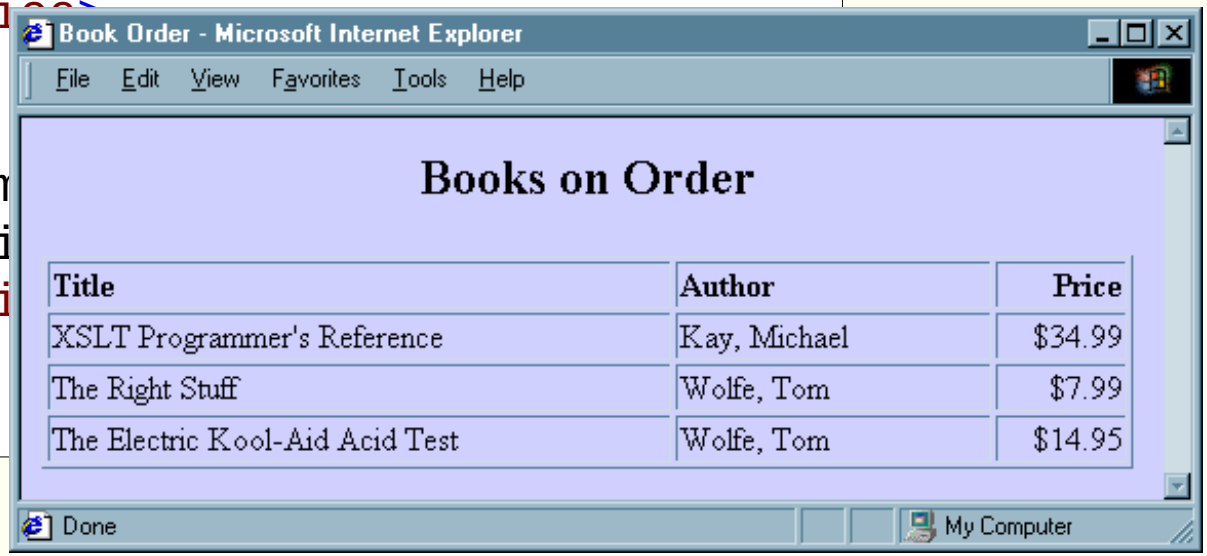
Repeating content in the middle

```
<xsl:template match="book">
  <!-- generate a table row for a book -->
  <tr>
    <td align="left" width="300">
      <xsl:value-of select="title"/>
    </td>
    <td align="left" width="150">
      <xsl:value-of select="author"/>
    </td>
    <td align="right" width="60">
      <xsl:value-of select="price"/>
    </td>
  </tr>
</xsl:template>
```

- Technique: template for each <book> element
  - generating one <tr> element and its contents
  - the <xsl:value-of> selects data from its child nodes

Stylesheet example: sorting

```
<?xml version="1.0" encoding="UTF-8"?>
<book-order>
  <book>
    <author>Wolfe, Tom</author>
    <title>The Right Stuff</title>
    <price>$7.99</price>
  </book>
  <book>
    <author>Kay, Michael</author>
    <title>XSLT Programmer's Reference</title>
    <price>$34.99</price>
  </book>
  <book>
    <author>Wolfe, Tom
    <title>The Electri
    <price>$14.95</pri
  </book>
</book-order>
```



xsl:sort element

- Purpose: sort nodes before processing

  without affecting the order of the nodes in the source tree

  - used inside <xsl:apply-templates> element
  - or used inside <xsl:for-each> element

- Example: sort by author name

```
<xsl:template match="book-order">
  <xsl:apply-templates>
    <xsl:sort select="author"/>
  </xsl:apply-templates>
</xsl:template>
```

- Example: sort by price and by author name

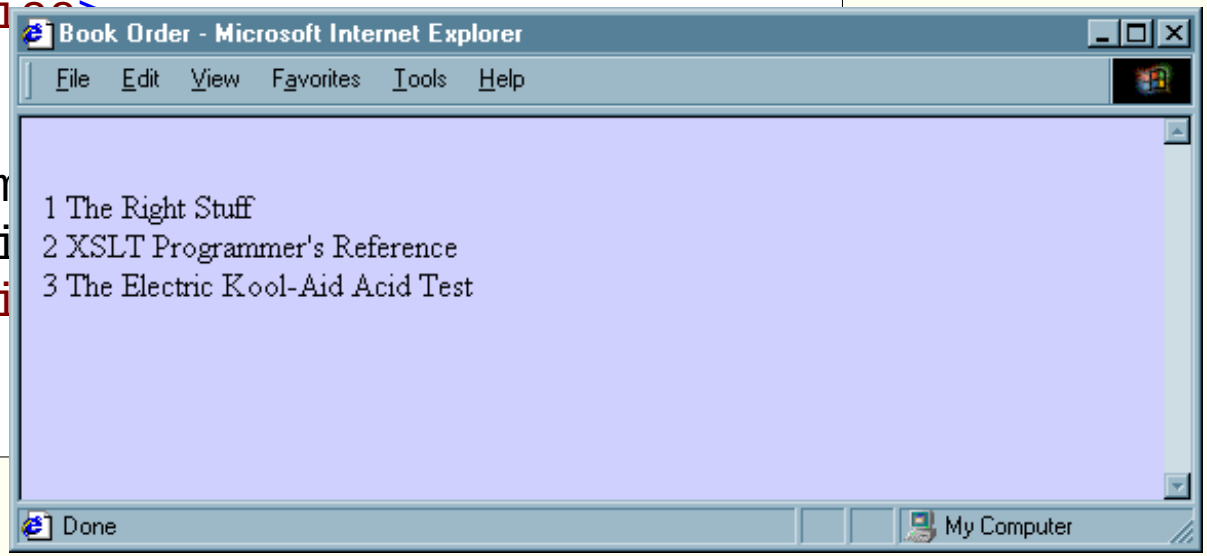
```
<xsl:sort select="price"
          order="descending"
          data-type="number"/>
<xsl:sort select="author"/>
```

sort order (default: ascending)

sort type (default: text)

Stylesheet example: numbering

```
<?xml version="1.0" encoding="UTF-8"?>
<book-order>
  <book>
    <author>Wolfe, Tom</author>
    <title>The Right Stuff</title>
    <price>$7.99</price>
  </book>
  <book>
    <author>Kay, Michael</author>
    <title>XSLT Programmer's Reference</title>
    <price>$34.99</price>
  </book>
  <book>
    <author>Wolfe, Tom
    <title>The Electri
    <price>$14.95</pri
  </book>
</book-order>
```



xsl:number element

- Purpose: give a sequence number to nodes
  by looking at the preceeding siblings of the node
- Example:

```
<xsl:template match="book">
  <xsl:number format="1"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="title"/><br/>
</xsl:template>
```

puts a blank between number and title

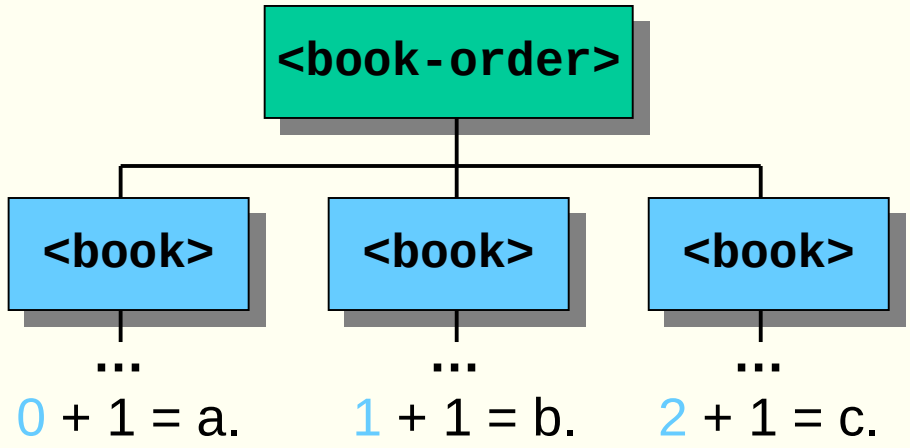
- Format:

"1"	"1."	"1)"	"a."	"A."	"I."	"(i)"
1 een 2 twee 3 drie	1. een 2. twee 3. drie	1) een 2) twee 3) drie	a. een b. twee c. drie	A. een B. twee C. drie	I. een II. twee III. drie	(i) een (ii) twee (iii) drie

xsl:number element

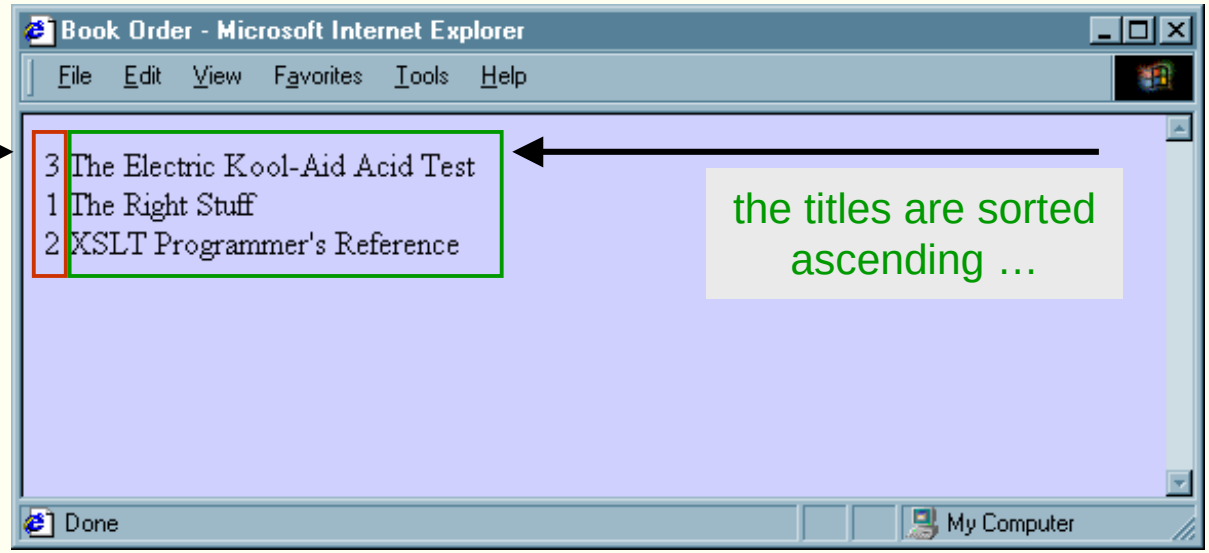
- How does it work?
  - counts number of preceding siblings of the element in the source tree, adds 1 to this number, and gives the result as text
- Example:

```
<xsl:template match="book">
  <xsl:number format="a."/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="title"/><br/>
</xsl:template>
```



Stylesheet example: sorting and numbering

```
<xsl:template match="book-order">
  <xsl:apply-templates>
    <xsl:sort select="title"/>
  </xsl:apply-templates>
</xsl:template>
<xsl:template match="book">
  <xsl:number/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="title"/><br/>
</xsl:template>
```



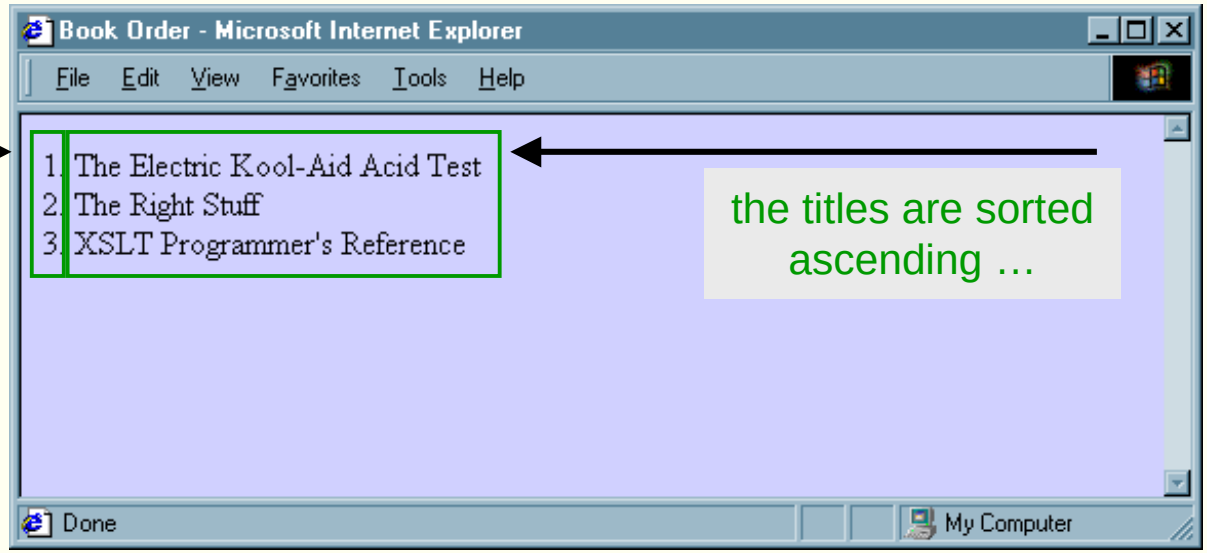
... but the numbers are wrong!

the titles are sorted ascending ...

Stylesheet example: sorting and numbering

```
<xsl:template match="book-order">
  <xsl:for-each select="book">
    <xsl:sort select="title"/>
    <xsl:number value="position()" format="1. "/>
    <xsl:value-of select="title"/><br/>
  </xsl:for-each>
</xsl:template>
```

… and also the numbers are right!



the titles are sorted ascending …

`xsl:for-each` element

- Purpose: loop through a set of nodes
  - by default sorted in document order (source tree)
  - can be used with `<xsl:sort>` element to specify sort order

  à `position()` reflects *sorted* order, not source *document* order

- Syntax:

```
<xsl:for-each select="node-set-expression">
   <!-- instructions: xsl:sort, xsl:template, … -->
</xsl:for-each>
```

  - the instructions are executed for each node that satisfies the `node-set-expression`

Stylesheet example: transforming XML à XML

XML input file

```
<?xml version="1.0" encoding="UTF-8"?>
<company>
  <div no="7a">
    <dept no="42">
      <emp no="123456" name="Whoptimone, Ida" role="sales"/>
      <emp no="651432" name="Tirebiter, George" role="sales"/>
    </dept>
    <dept no="51">
      <emp no="832953" name="Danger, Nick" role="market"/>
    </dept>
  </div>
  <div no="2b">
    <dept no="57">
      <emp no="283412" name="Boss, Yuda" role="admin"/>
      <emp no="283412" name="Gates, Billy" role="boss"/>
    </dept>
  </div>
</company>
```

Stylesheet example: transforming XML à XML

XML output file (desired)

```
<?xml version="1.0" encoding="UTF-8"?>
<company>
  <employee id="123456">
    <name>Whoptimone, Ida</name>
    <division>7a</division><department>42</department>
  </employee>
  <employee id="651432">
    <name>Tirebiter, George</name>
    <division>7a</division><department>42</department>
  </employee>
  <employee id="832953">
    <name>Danger, Nick</name>
    <division>7a</division><department>51</department>
  </employee>
  <employee id="283412">
    <name>Boss, Yuda</name>
    <division>2b</division><department>57</department>
  </employee>
</company>
```

Stylesheet example: transforming XML à XML

- What changed?
  - the structure of the document
    - was a company directory organised by division and department
    - is now a company directory organised by employee
  - the name of elements and attributes
  - most values in attributes are now in element contents
  - the information about the boss has been deleted from the document
- Note: the sequencing of information in the source tree (how attribute values and element contents follow each other in the document) hasn't really changed significantly
  - impossible to change using declarative programming
  - can be changed using procedural programming, but it's hard!

xsl:element element

- Purpose: create a node in the result tree
  - content of the node =
    - attributes (created using <xsl:attribute>)
    - child elements (created using <xsl:element>)
- Syntax:

```
<xsl:element name="element-name">
  <!-- content: attributes, child elements -->
</xsl:element>
```

  - a node with name element-name is created with the specified content (of attributes and child elements)
- Alternative syntax:

```
<element-name>
  <!-- content: attributes, child elements -->
</element-name>
```

xsl:attribute element

- Purpose: create an attribute for a node in the result tree
  - value of the attribute = text value of the content of `<xsl:attribute>`
- Syntax:

```
<xsl:attribute name="attribute-name">
  <!-- content: text value -->
</xsl:attribute>
```

  - the text value is given to the attribute with name `attribute-name`
- Example:

```
<xsl:attribute name="id">
  <xsl:value-of select="@no"/>
</xsl:attribute>
```

  - create an attribute named `id`
  - give it the value of the attribute `no` of the current node

Stylesheet example: transforming XML à XML

```
<xsl:template match="/">
  <xsl:element name="company">
    <xsl:apply-templates select="company/*/*/emp"/>
  </xsl:element>
</xsl:template>

<xsl:template match="emp">
  <xsl:element name="employee">
    <xsl:attribute name="id">
      <xsl:value-of select="@no"/>
    </xsl:attribute>
    <name>
      <xsl:value-of select="@name"/>
    </name>
    <xsl:element name="division">
      <xsl:value-of select="../../@no"/>
    </xsl:element>
    <department>
      <xsl:value-of select="../@no"/>
    </department>
  </xsl:element>
</xsl:template>
```

Let's see if the stylesheet works!

XML output file

```
<company>
  <employee id="123456">
    <name>Whoptimone, Ida</name>
    <division>7a</division>
    <department>42</department>
  </employee>
  ...
  <employee id="283412">
    <name>Boss, Yuda</name>
    <division>2b</division>
    <department>57</department>
  </employee>
  <employee id="283412">
    <name>Gates, Billy</name>
    <division>2b</division>
    <department>57</department>
  </employee>
</company>
```

What's the boss still doing here?

Stylesheet example: transforming XML à XML

```
<xsl:template match="/">
  <xsl:element name="company">
    <xsl:apply-templates select="company/*/*/emp"/>
  </xsl:element>
</xsl:template>

<xsl:template match="emp">
  <xsl:if test="not(@role='boss')">
    <xsl:element name="employee">
      <xsl:attribute name="id">
        <xsl:value-of select="@no"/>
      </xsl:attribute>
      <name>
        <xsl:value-of select="@name"/>
      </name>
      <xsl:element name="division">
        <xsl:value-of select="../../@no"/>
      </xsl:element>
      ...
    </xsl:element>
  </xsl:if>
</xsl:template>
```

xsl:if element

- Purpose: conditional execution of a single option
  - instructions to be executed = the content of <xsl:if>
- Syntax:

```
<xsl:if test="boolean-expression">
  <!-- instructions: literal result text, XSLT elements -->
</xsl:if>
```

  - the test condition boolean-expression is an XPath expression that results in a Boolean value
- Example:

```
<xsl:if test="@priority='3'">
  Priority: high
</xsl:if>
```

xsl:choose element

- Purpose: conditional execution of multiple options
- Syntax:

```
<xsl:choose>
  <xsl:when test="boolean-expression">
   <!-- instructions: literal result text, XSLT elements -->
  </xsl:when>
  ...
  <xsl:otherwise>
   <!-- instructions: literal result text, XSLT elements -->
  </xsl:otherwise>
</xsl:choose>
```

- Example:

```
<xsl:choose>
  <xsl:when test="@priority='3'">Priority: high</xsl:when>
  <xsl:when test="@priority='2'">Priority: medium</xsl:when>
  <xsl:otherwise>Priority: low</xsl:otherwise>
</xsl:choose>
```

Stylesheet example: transforming XML à XML

XML output file (desired)

```
<?xml version="1.0" encoding="UTF-8"?>
<company>
  <ids>
    <id>123456</id><id>651432</id><id>832953</id><id>283412</id>
  </ids>
  <employee id="123456">
    <name>Whoptimone, Ida</name>
    <division>7a</division><department>42</department>
  </employee>
  <employee id="651432">
    <name>Tirebiter, George</name>
    <division>7a</division><department>42</department>
  </employee>
  <employee id="832953">
    <name>Danger, Nick</name>
    <division>7a</division><department>51</department>
  </employee>
  <employee id="283412">
    <name>Boss, Yuda</name>
    <division>2b</division><department>57</department>
  </employee>
</company>
```

Stylesheet example: transforming XML à XML

```
<xsl:template match="/">
  <xsl:element name="company">
    <ids>
      <xsl:apply-templates select="company/*/*/emp" mode="get-ids"/>
    </ids>
    <xsl:apply-templates select="company/*/*/emp"/>
  </xsl:element>
</xsl:template>

<xsl:template match="emp" mode="get-ids">
  <xsl:if test="not(@role='boss')">
    <id>
      <xsl:value-of select="@no"/>
    </id>
  </xsl:if>
</xsl:template>

<xsl:template match="emp">
  <xsl:if test="not(@role='boss')">
    <xsl:element name="employee">
    ...
    </xsl:element>
  </xsl:if>
</xsl:template>
```

Using modes

- Collections of template rules can be made:

```
<xsl:apply-templates select="match-expr" mode="mode-name"/>
```

  - to indicate that the collection `mode-name` has to be used

```
<xsl:template select="match-expr" mode="mode-name">
  <!-- instructions: literal result text, XSLT elements -->
</xsl:template>
```

  - to indicate that the template belongs to the collection `mode-name`
  - when `mode="…"` is not specified, the unnamed collection is assumed

- Advantages of using modes
  - modes can be changed at any time
  - there can be any number of different modes
    each reflecting different requirements for the stylesheet
  - by grouping templates together in collections
    each collection can be treated as a separate stylesheet

Two models of XSLT processing

- "**Push**" model, typically used for displaying documents
  - source data structure is unknown or arbitrary
  - output structure has to follow input structure
  - templates for various elements in source tree
  - same approach as in most style languages (e.g. CSS)
- "**Pull**" model, typically used for restructuring data
  - source data structure is known, just "fill-in-the-blanks"
  - output structure does not have to follow input structure
  - template for root, use `<xsl:for-each>` to retrieve elements of interest, use `<xsl:value-of>` for particular elements/attributes to output data
  - same approach as in server pages (e.g. ASP, JSP)
- XSLT allows *both* in the same stylesheet
  - *pushing* through templates, *pulling* from within a template

Tips on developing good XSLT stylesheets

- For display format generation:
  - create a working HTML/WML/… prototype
  - copy parts of the prototype into the stylesheet
  - build the stylesheet one single template at a time
- For XML vocabulary translation:
  - make sure you know the exact meaning and format for each element in the source and result XML vocabularies
  - determine how source elements/attributes need to be mapped in result elements/attributes
  - make a list of data not present in the source document that needs to be generated, and of data to be deleted from the target document
- If XSLT by itself is not sufficient, consider using extensions
  - using built-in XSLT processor-specific extension tags
  - calling user-defined extension functions (Java, VBScript,…)

Tips on achieving better XSLT performance

- Keep the source document small,
  if necessary split the document in smaller documents first
- Keep the result document small,
  e.g if you're generating HTML use a CSS stylesheet to style it
- Split complex transformations into several stages
- Keep the patterns in template rules simple
  - use `/descendant::*` in preference to `//*`
  - avoid using positional predicates in patterns, e.g. `para[last()-1]`
  - use `<xsl:choose>` conditional logic within the template rather than having dozens of different template rules to be matched
- Use `position()` in preference to `<xsl:number/>`
- To output the content of a text-only element, use `<xsl:value-of>` in preference to `<xsl:apply-templates>`

Tips on achieving better XSLT performance

- If the previous tips alone are not sufficient, consider finetuning
  - using the server-side stylesheet caching features of MSXML 3.0

    http://msdn.microsoft.com/library/psdk/xmlsdk/xmls8wc3.htm
    - use the XSLTemplate object to cache stylesheet
  - compiling XSL stylesheets to Java "translets"
    using the technology preview *XSLT Compiler* from Sun

    http://www.sun.com/xml/developers/xsltc/
  - compiling XSL stylesheets to optimized machine code
    using *XSLJIT*, the Just-In-Time Compiler for XSL from DataPower

    http://www.datapower.com/technology.shtml#XSLJIT
  - using a complete XSL publishing framework
    like Apache's *Cocoon*

    http://xml.apache.org/cocoon/
    - precompiled dynamic page generation
    - caching of dynamically generated pages