XSLT

Patryk Czarnik

Institute of Informatics University of Warsaw

XML and Modern Techniques of Content Management – 2010/11
XSLT Features

- Idea
- Control flow instructions
- Sorting and grouping
- Result construction
- Variables and parameters
- Named templates and functions
- Serialisation methods
- Applications
XSLT Features

- Idea
- Control flow instructions
- Sorting and grouping
- Result construction
- Variables and parameters
- Named templates and functions
- Serialisation methods
- Applications
XSLT — status

• Created within XSL.
• Applications go beyond XML visualisation.
• Version 1.0:
  • October 1999, related to XPath 1.0
  • good support in tools and libraries
• Version 2.0:
  • January 2007, connected with XPath 2.0 and XQuery 1.0,
  • new features
  • less software support
XSLT — status

- Created within XSL.
- Applications go beyond XML visualisation.
  - Version 1.0:
    - October 1999, related to XPath 1.0
    - good support in tools and libraries
  - Version 2.0:
    - January 2007, connected with XPath 2.0 and XQuery 1.0,
    - new features
    - less software support
Created within XSL.

Applications go beyond XML visualisation.

Version 1.0:
- October 1999, related to XPath 1.0
- good support in tools and libraries

Version 2.0:
- January 2007, connected with XPath 2.0 and XQuery 1.0,
- new features
- less software support
Created within XSL.

Applications go beyond XML visualisation.

Version 1.0:
- October 1999, related to XPath 1.0
- good support in tools and libraries

Version 2.0:
- January 2007, connected with XPath 2.0 and XQuery 1.0,
- new features
- less software support
XSLT — support

- XSLT 2.0 processors:
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- XSLT 1.0 processors:
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - ...

- Authoring tools
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — support

- XSLT 2.0 processors:
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial schema aware version
  - XML Spy (commercial windows application)

- XSLT 1.0 processors:
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - ...

- Authoring tools
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen)
XSLT — support

- **XSLT 2.0 processors:**
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)
- **XSLT 1.0 processors:**
  - internet browsers
    - Xalan (Java and C++ libraries)
    - xsltproc, part of libxml (C, basically for Linux)
    - XML extensions of database engines
    - ...
- **Authoring tools**
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — support

- XSLT 2.0 processors:
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial schema aware version
  - XML Spy (commercial windows application)

- XSLT 1.0 processors:
  - internet browsers
  - Xalan (Java and C++ libraries)
    - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - …

- Authoring tools
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — support

- XSLT 2.0 processors:
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- XSLT 1.0 processors:
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - …

- Authoring tools
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen)
XSLT — support

- **XSLT 2.0 processors:**
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- **XSLT 1.0 processors:**
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - ...

- **Authoring tools**
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen)
XSLT — support

- **XSLT 2.0 processors:**
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- **XSLT 1.0 processors:**
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - ...

- **Authoring tools**
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — support

- **XSLT 2.0 processors:**
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- **XSLT 1.0 processors:**
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - …

- **Authoring tools**
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — support

- **XSLT 2.0 processors:**
  - Saxon
    - Java and .NET libraries, command-line applications
    - free (Open Source) basic version
    - commercial *schema aware* version
  - XML Spy (commercial windows application)

- **XSLT 1.0 processors:**
  - internet browsers
  - Xalan (Java and C++ libraries)
  - xsltproc, part of libxml (C, basically for Linux)
  - XML extensions of database engines
  - …

- **Authoring tools**
  - raw text editors and developer environments (e.g. Eclipse)
  - specialised tools — rather paid (e.g. XML Spy, oXygen).
XSLT — stylesheet structure

- **Stylesheet** (*arkusz*) consists of templates.
- **Template** (*szablon*) specifies how to transform a source node into result tree fragment.
- Content of template — **sequence constructor**:
  - text and elements not from XSLT namespace → copied to result
  - XSLT instructions → affects processing
  - XPath in instructions → access to source document, arithmetic, conditions testing, . . .
- XSLT can be considered a programming language specialised for XML documents transformation.
XSLT — stylesheet structure

- **Stylesheet** (*arkusz*) consists of templates.
- **Template** (*szablon*) specifies how to transform a source node into result tree fragment
  - Content of template — *sequence constructor*:
    - text and elements not from XSLT namespace → copied to result
    - XSLT instructions → affects processing
    - XPath in instructions → access to source document, arithmetic, conditions testing, . . .
  - XSLT can be considered a programming language specialised for XML documents transformation.
XSLT — stylesheet structure

- **Stylesheet** *(arkusz)* consists of templates.
- **Template** *(szablon)* specifies how to transform a source node into result tree fragment.
- Content of template — **sequence constructor**:
  - text and elements not from XSLT namespace → copied to result
  - XSLT instructions → affects processing
  - XPath in instructions → access to source document, arithmetic, conditions testing, . . .

XSLT can be considered a programming language specialised for XML documents transformation.
XSLT — stylesheet structure

- **Stylesheet** (*arkusz*) consists of templates.
- **Template** (*szablon*) specifies how to transform a source node into result tree fragment.
- Content of template — **sequence constructor**:
  - text and elements not from XSLT namespace → copied to result
  - XSLT instructions → affects processing
  - XPath in instructions → access to source document, arithmetic, conditions testing, . . .
- XSLT can be considered a programming language specialised for XML documents transformation.
XSLT — stylesheet structure

- **Stylesheet** (*arkusz*) consists of templates.
- **Template** (*szablon*) specifies how to transform a source node into result tree fragment.
- Content of template — **sequence constructor**:
  - text and elements not from XSLT namespace → copied to result
  - XSLT instructions → affects processing
  - XPath in instructions → access to source document, arithmetic, conditions testing, ...

XSLT can be considered a programming language specialised for XML documents transformation.
**XSLT — stylesheet structure**

- **Stylesheet** (*arkusz*) consists of templates.
- **Template** (*szablon*) specifies how to transform a source node into result tree fragment.
- Content of template — **sequence constructor**: 
  - text and elements not from XSLT namespace → copied to result
  - XSLT instructions → affects processing
  - XPath in instructions → access to source document, arithmetic, conditions testing, ...
- XSLT can be considered a programming language specialised for XML documents transformation.
Stylesheet structure — example

- main element

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

    <xsl:output method="html" encoding="iso-8859-2" />
    <xsl:import href="other_stylesheet.xsl"/>
    <xsl:param name="css"/>

    <xsl:template match="/">
        <html>
            <head><link rel="stylesheet" type="text/css" href="${css}" /></head>
            <body><xsl:apply-templates /></body>
        </html>
    </xsl:template>

    <xsl:template match="paragraph">
        <p><xsl:apply-templates select="." /></p>
    </xsl:template>

</xsl:stylesheet>
```
Stylesheet structure — example

- declarations

```xml
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" encoding="iso-8859-2" />
    <xsl:import href="other_stylesheet.xsl"/>
    <xsl:param name="css"/>

    <xsl:template match="/">
        <html>
            <head><link rel="stylesheet" type="text/css" href="{$css}"/></head>
            <body><xsl:apply-templates /></body>
        </html>
    </xsl:template>

    <xsl:template match="paragraph">
        <p><xsl:apply-templates select="." /></p>
    </xsl:template>

</xsl:stylesheet>
```
Stylesheet structure — example

templates

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

    <xsl:output method="html" encoding="iso-8859-2" />
    <xsl:import href="other_stylesheet.xsl"/>
    <xsl:param name="css"/>

    <xsl:template match="/">
        <html>
            <head><link rel="stylesheet" type="text/css" href="{$css}"/></head>
            <body><xsl:apply-templates /></body>
        </html>
    </xsl:template>

    <xsl:template match="paragraph">
        <p><xsl:apply-templates select="." /></p>
    </xsl:template>

</xsl:stylesheet>
```
Stylesheet structure — example

“sequence constructor”

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

    <xsl:output method="html" encoding="iso-8859-2" />
    <xsl:import href="other_stylesheet.xsl"/>
    <xsl:param name="css"/>

    <xsl:template match="/">
        <html>
            <head><link rel="stylesheet" type="text/css" href="{$css}"/></head>
            <body><xsl:apply-templates /></body>
        </html>
    </xsl:template>

    <xsl:template match="paragraph">
        <p><xsl:apply-templates select="." /></p>
    </xsl:template>

</xsl:stylesheet>
```
XSLT — operation overview

○ Transformation at tree level
  ○ Template for document node (root) started first
    ○ template exists even if we had not written it
  ○ apply-templates within a template — another templates called for other nodes, usually for children
  ○ Templates matched according to node kind and name, location within document tree, etc.
  ○ Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
XSLT — operation overview

- Transformation at tree level
- Template for document node (root) started first
  - template exists even if we had not written it
- `apply-templates` within a template — another templates called for other nodes, usually for children
- Templates matched according to node kind and name, location within document tree, etc.
- Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
XSLT — operation overview

- Transformation at tree level
- Template for document node (root) started first
  - template exists even if we had not written it
- `apply-templates` within a template — another templates called for other nodes, usually for children
- Templates matched according to node kind and name, location within document tree, etc.
- Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
XSLT — operation overview

- Transformation at tree level
- Template for document node (root) started first
  - template exists even if we had not written it
- `apply-templates` within a template — another templates called for other nodes, usually for children
- Templates matched according to node kind and name, location within document tree, etc.
- Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
XSLT — operation overview

- Transformation at tree level
- Template for document node (root) started first
  - template exists even if we had not written it
- `apply-templates` within a template — another templates called for other nodes, usually for children
- Templates matched according to node kind and name, location within document tree, etc.

- Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
XSLT — operation overview

- Transformation at tree level
- Template for document node (root) started first
  - template exists even if we had not written it
- apply-templates within a template — another templates called for other nodes, usually for children
- Templates matched according to node kind and name, location within document tree, etc.
- Additional features: conditional processing, copying values and nodes from source, generating new nodes, grouping, sorting, ...
Templates

**template**
- transformation of one node into result tree fragment
- `match` — to which nodes the template applies

**apply-templates**
- applying templates matching (individually) given nodes
- `select` — nodes to process (children by default)

```xml
<xsl:template match="department">
  <ul>
    <xsl:apply-templates select="person"/>
  </ul>
</xsl:template>

<xsl:template match="person">
  <li><xsl:apply-templates /></li>
</xsl:template>
```
Template selection

Patterns

- In `match` attributes of templates
- Restricted form of XPath

Finding appropriate template

- Node must “match” pattern
- When node matches many templates:
  - the most detailed `match` wins (details in Recommendation)
  - imports affect template priorities
  - possibility to explicitly give priority
  - conflict — error signalled or later template chosen (depends on implementation)
Template selection

Patterns
- In `match` attributes of templates
- Restricted form of XPath

Finding appropriate template
- Node must “match” pattern
- When node matches many templates:
  - the most detailed `match` wins (details in Recommendation)
  - imports affect template priorities
  - possibility to explicitly give `priority`
  - conflict — error signalled or later template chosen (depends on implementation)
Processing modes

<xsl:template match="department">
    <table>
        <xsl:apply-templates select="person" mode="table"/>
    </table>
</xsl:template>

<xsl:template match="person">
    <li>
        <xsl:apply-templates select="name | surname"/>
    </li>
</xsl:template>

<xsl:template match="person" mode="table">
    <tr>
        <td>
            <xsl:value-of select="name"/>
        </td>
        <td>
            <xsl:value-of select="surname"/>
        </td>
    </tr>
</xsl:template>

Available values of mode:

- name
- #default
- #current — (on application)
- #all — (in template)
Processing modes

<xsl:template match="department">
  <table> ...
    <xsl:apply-templates select="person" mode="table"/>
  </table>
</xsl:template>

<xsl:template match="person">
  <li><xsl:apply-templates select="name | surname"/></li>
</xsl:template>

<xsl:template match="person" mode="table">
  <tr><td><xsl:value-of select="name"/></td>
    <td><xsl:value-of select="surname"/></td></tr>
</xsl:template>

Available values of mode

- name
- #default
- current — (on application)
- all — (in template)
Built-in templates

Applied when node matches no template provided by author.

Document node and elements
- apply templates for children (not for attributes!)
- pass all template parameters
- preserve current mode

Attributes and text nodes
- copy text value to result (insert text node)

Processing instructions and comments
- do nothing
Built-in templates

Applied when node matches no template provided by author.

**Document node and elements**
- apply templates for children (not for attributes!)
- pass all template parameters
- preserve current mode

**Attributes and text nodes**
- copy text value to result (insert text node)

**Processing instructions and comments**
- do nothing
Built-in templates

Applied when node matches no template provided by author.

**Document node and elements**

- apply templates for children (not for attributes!)
- pass all template parameters
- preserve current mode

**Attributes and text nodes**

- copy text value to result (insert text node)

**Processing instructions and comments**

- do nothing
### Built-in templates

Applied when node matches no template provided by author.

<table>
<thead>
<tr>
<th><strong>Document node and elements</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• apply templates for children (not for attributes!)</td>
</tr>
<tr>
<td>• pass all template parameters</td>
</tr>
<tr>
<td>• preserve currents mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Attributes and text nodes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• copy text value to result (insert text node)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Processing instructions and comments</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• do nothing</td>
</tr>
</tbody>
</table>
Built-in templates

**Document node and elements**

```
<xsl:template match="element()|document()" mode="#all">
    <xsl:param .../> ...
    <xsl:apply-templates select="child::node()" mode="#current">
        <xsl:with-param .../> ...
    </xsl:apply-templates>
</xsl:template>
```

**Attributes and text nodes**

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

**Processing instructions and comments**

```
<xsl:template match="processing-instruction()|comment()" mode="#all"/>
```
Built-in templates

Document node and elements

```xml
<xsl:template match="element()|document()" mode="#all">
    <xsl:param .../> ...
    <xsl:apply-templates select="child::node()" mode="#current">
        <xsl:with-param .../> ...
    </xsl:apply-templates>
</xsl:template>
```

Attributes and text nodes

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

Processing instructions and comments

```xml
<xsl:template match="processing-instruction()|comment()" mode="#all"/>
```
Built-in templates

**Document node and elements**

```xml
<xsl:template match="element()|document()" mode="#all">
    <xsl:param .../> ...
    <xsl:apply-templates select="child::node()" mode="#current">
        <xsl:with-param .../> ...
    </xsl:apply-templates>
</xsl:template>
```

**Attributes and text nodes**

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

**Processing instructions and comments**

```xml
<xsl:template match="processing-instruction()|comment()" mode="#all"/>
```
**Sequence iteration**

**Instruction** `for-each`

- Iteration over all items selected by `select`.
- XSLT 1.0 — iteration over a node set
- XSLT 2.0 — iteration over any sequence

**Example**

```xml
<xsl:template match="department">
  <ul>
    <xsl:for-each select="person">
      <li><xsl:value-of select="name"/></li>
      <li><xsl:value-of select="surname"/></li>
    </xsl:for-each>
  </ul>
</xsl:template>
```
Conditional processing — single (optional) branch

Instruction \textbf{if}

- Testing logical condition from \texttt{test} \textit{(Effective Boolean Value)}
- If true, processing internals and inserting result
- No \texttt{else} here

Example

```xml
<xsl:for-each match="product">
  <xsl:apply-templates />
  <xsl:if test="position() != last()">
    <xsl:text>, </xsl:text>
  </xsl:if>
</xsl:template>
```
**Conditional processing — many branches**

**Instruction** `choose`

- Conditions *(test)* evaluated in the given order
- One branch chosen (the first with satisfied condition)
- Optional *otherwise* branch

**Example**

```xml
<xsl:template match="account">
Account balance is
   <xsl:choose>
      <xsl:when test="saldo &gt; 0">positive</xsl:when>
      <xsl:when test="saldo &lt; 0">negative</xsl:when>
      <xsl:otherwise>equal to zero</xsl:otherwise>
   </xsl:choose>.
</xsl:template>
```
Sorting during processing

Instruction **sort**

- May appear within *for-each*, *for-each-group* or *apply-templates*
- Sorting options in attributes:
  - `select` sorting key
  - `data-type` *(text/number)*
  - `order`, `case-order`, `stable`, `lang`

**Example**

```xml
<xsl:template match="exam-results">
  <ul>
    <xsl:apply-templates select="student">
      <xsl:sort select="points" data-type="number" order="descending"/>
      <xsl:sort select="surname" data-type="text"/>
    </xsl:apply-templates>
  </ul>
</xsl:template>
```
Sorting any sequences

Instruction **perform-sort**

Example from recommendation which illustrates also custom function definition

```xml
<xsl:function name="bib:books-by-price" as="schema-element(bib:book)\*">
  <xsl:param name="in" as="schema-element(bib:book)\*"/>
  <xsl:perform-sort select="\$in">
    <xsl:sort select="xs:decimal(bib:price)"/>
  </xsl:perform-sort>
</xsl:function>

... 
  [position() = 1 to 5]"/>
```
Result tree nodes

- Written in XSLT stylesheets
  - convenient
- Nodes created with instructions (<xsl:element>, <xsl:comment> etc.)
  - general
- Nodes copied from source document
Direct construction of result nodes

Nodes copied from stylesheet to result

- Elements not from XSLT namespace
  - with attributes
  - content processed as sequence constructor, result put into constructed element content

- Text nodes containing some “black” characters

```xml
<xsl:template match="chapter">
  <div>
    Chapter
    <xsl:apply-templates />
  </div>
</xsl:template>
```
Instructions that create nodes

- For each kind of node “constructor” instruction.

```xml
<xsl:document>
  <xsl:processing-instruction target="xml-stylesheet">
    type="text/css" href="styl.css"
  </xsl:processing-instruction>

  <xsl:element name="p">
    <xsl:attribute name="class">abstract</xsl:attribute>
    <xsl:text>This article is about this and that.</xsl:text>
  </xsl:element>

  <xsl:comment>This will be a comment.</xsl:comment>
</xsl:document>
```
Constructor instructions — practical applications

- To insert a comment or a processing instruction.
- To insert only white characters.
- To insert a text *without* unneeded white characters.
- To establish element (or attribute) name dynamically.
- To insert an attribute conditionally.

Example

```xml
<xsl:processing-instruction target="xml-stylesheet">
    type="text/css" href="styl.css"
</xsl:processing-instruction>

<xsl:comment>Modification date:
    <xsl:value-of select="current-date()"/></xsl:comment>
```
Constructor instructions — practical applications

- To insert a comment or a processing instruction.
- To insert only white characters.
- To insert a text without unneeded white characters.
- To establish element (or attribute) name dynamically.
- To insert an attribute conditionally.

Example

```xml
<xsl:for-each select="person">
  <xsl:value-of select="@email"/>
  <xsl:if test="position() != last()">
    <xsl:text> </xsl:text>
  </xsl:if>
</xsl:for-each>
```
Constructor instructions — practical applications

- To insert a comment or a processing instruction.
- To insert only white characters.
- To insert a text *without* unneeded white characters.
- To establish element (or attribute) name dynamically.
- To insert an attribute conditionally.

**Example**

```
<xsl:for-each select="person">
  <xsl:value-of select="@email"/>
  <xsl:if test="position() != last()"/>
    <xsl:text>, </xsl:text>
  </xsl:if>
</xsl:for-each>
```
Constructor instructions — practical applications

- To insert a comment or a processing instruction.
- To insert only white characters.
- To insert a text *without* unneeded white characters.
- To establish element (or attribute) name dynamically.
- To insert an attribute conditionally.

**Example**

```xml
<xsl:element name="h{min((count(ancestor-or-self::section), 6))}">
  ...
</xsl:element>
```
Constructor instructions — practical applications

- To insert a comment or a processing instruction.
- To insert only white characters.
- To insert a text *without* unneeded white characters.
- To establish element (or attribute) name dynamically.
- To insert an attribute conditionally.

Example

```xml
<p>
  <xsl:if test="@position = 'manager'">
    <xsl:attribute name="class">
      mgr
    </xsl:attribute>
  </xsl:if>
</p>
```
Attribute sets

Definition example

```xml
<xsl:attribute-set name="important">
  <xsl:attribute name="font-weight">bold</xsl:attribute>
  <xsl:attribute name="color">red</xsl:attribute>
</xsl:attribute-set>
```

Use example

```xml
<xsl:template match="emph">
  <fo:inline xsl:use-attribute-sets="important">
    <xsl:apply-templates />
  </fo:inline>
</xsl:template>

<xsl:template match="remark">
  <fo:block xsl:use-attribute-sets="important">
    <xsl:apply-templates />
  </fo:block>
</xsl:template>
```
Attribute sets

**Definition example**

```
<xsl:attribute-set name="important">
  <xsl:attribute name="font-weight">bold</xsl:attribute>
  <xsl:attribute name="color">red</xsl:attribute>
</xsl:attribute-set>
```

**Use example**

```
<xsl:template match="emph">
  <fo:inline xsl:use-attribute-sets="important">
    <xsl:apply-templates />
  </fo:inline>
</xsl:template>

<xsl:template match="remark">
  <fo:block xsl:use-attribute-sets="important">
    <xsl:apply-templates />
  </fo:block>
</xsl:template>
```
Inserting XPath result

- **XSLT instructions** `sequence`, `copy-of` and `value-of`.
- **XPath expression in attribute** `select`
  - or sequence constructor inside for `value-of` in XSLT 2.0

- Inserted to result:
  - `sequence` computed sequence
  - `copy-of` (deep) copy of sequence
  - `value-of` text node with string value of sequence

...differences between XSLT 1.0 and 2.0!
XSLT Features

XPath result insertion — examples

```xml
<xsl:sequence select="for $i in (1 to 10) return $i * $i"/>

<xsl:sequence select="/employee[salary > 2000]"/>

<xsl:copy-of select="/employee[salary > 2000]"/>

<xsl:value-of select="/employee[salary > 2000]/surname"/>

<xsl:value-of>
  <xsl:apply-templates select="employee"/>
</xsl:value-of>
```
**value-of in XSLT 1.0**

Only the first node of selected nodes set is taken and result of casting it to string is inserted to the result as a text node.

**Document**

```
<person><name>Patryk</name><name>Czarnik</name></person>
<person><name>Maciej</name><name>Ogrodniczuk</name></person>
```

**Stylesheet**

```
<result><xsl:value-of select="//person/name"/></result>
```

**Result**

```
<result>Patryk</result>
```
value-of in XSLT 2.0

The sequence is atomized and each atom is casted to string. Then all strings are inserted to the result, separated with a space (or another separator given in separator attribute).

Document

```
<person><name>Patryk</name><name>Czarnik</name></person>
<person><name>Maciej</name><name>Ogrodniczuk</name></person>
```

Stylesheet

```
<result><xsl:value-of select="//person/name"/></result>
```

Result

```
<result>Patryk Maciej</result>
```
Attribute value templates
Convenient way to dynamically compute attribute values

- To be used in
  - attributes of literal result elements
  - some attributes of XSLT instructions

- Fixed parts — strings copied to result
  - { and } should be written as {{ and }}

- Variable parts — evaluated dynamically
  - XPath expression between { and }
  - text representation of evaluated value inserted
  - the same rules as for value-of, space is the separator
  - the same inconsistency between XSLT 1.0 and XSLT 2.0

```xml
<img src="{$image_server}/@id.jpg"/>

<xsl:element name="n{count(ancestor-or-self::section)}">
  ...
</xsl:element>
```
**Attribute value templates**

Convenient way to dynamically compute attribute values

- To be used in
  - attributes of literal result elements
  - some attributes of XSLT instructions

- **Fixed parts** — strings copied to result
  - \{ and \} should be written as \{{ and \}\

- **Variable parts** — evaluated dynamically
  - XPath expression between \{ and \}
  - text representation of evaluated value inserted
  - the same rules as for value-of, space is the separator
  - the same inconsistency between XSLT 1.0 and XSLT 2.0

```xml
<xsl:element name="h{count(ancestor-or-self::section)}">
  ...
</xsl:element>
```
Attribute value templates

Convenient way to dynamically compute attribute values

- To be used in
  - attributes of literal result elements
  - some attributes of XSLT instructions

- Fixed parts — strings copied to result
  - `{ and }` should be written as `{ { and } }

- Variable parts — evaluated dynamically
  - XPath expression between `{ and }`
  - text representation of evaluated value inserted
  - the same rules as for `value-of`, space is the separator
  - the same inconsistency between XSLT 1.0 and XSLT 2.0

```xml
<img src="{$image_server}/@id}.jpg"/>

<xsl:element name="h{count(ancestor-or-self::section)}">
  ...
</xsl:element>
```
Local variables

- XSLT variables and params → XPath variables
- “Declarative” variables — no assignment instruction

Example

```xml
<xsl:template match="account">
  <xsl:variable name="sign">
    <xsl:choose>
      <xsl:when test="balance &gt; 0">positive</xsl:when>
      <xsl:when test="balance &lt; 0">negative</xsl:when>
      <xsl:otherwise>equal to zero</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  Account balance is <xsl:value-of select="$sign"/>.
</xsl:template>
```
Consequences of variables declarative nature

Variable not defined in the place of reference

```xml
<xsl:choose>
  <xsl:when test="balance &ge; 0">
    <xsl:variable name="sign">non-negative</xsl:variable>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="sign">negative</xsl:variable>
  </xsl:otherwise>
</xsl:choose>

Account balance is <xsl:value-of select="$sign"/>.
```

New variable covers old variable (not permitted by some processors)

```xml
<xsl:variable name="sign">non-negative</xsl:variable>
<xsl:if test="balance &lt; 0">
  <xsl:variable name="sign">negative</xsl:variable>
</xsl:if>

Account balance is <xsl:value-of select="$sign"/>.
```
Consequences of variables declarative nature

Variable not defined in the place of reference

```xml
<xsl:choose>
  <xsl:when test="balance &ge; 0">
    <xsl:variable name="sign">non-negative</xsl:variable>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="sign">negative</xsl:variable>
  </xsl:otherwise>
</xsl:choose>

Account balance is <xsl:value-of select="$sign"/>.
```

New variable covers old variable (not permitted by some processors)

```xml
<xsl:variable name="sign">non-negative</xsl:variable>
<xsl:if test="balance &lt; 0">
  <xsl:variable name="sign">negative</xsl:variable>
</xsl:if>

Account balance is <xsl:value-of select="$sign"/>.
```
Global parameters and variables

- Visible in the whole stylesheet (unless covered by local variables)
- Parameter values provided “from outside”
- Variable values computed once at transformation start

Example

```xml
<xsl:param name="name"/>
<xsl:variable name="how-many-elements"
    select="count(/*[local-name() = $name])"/>

<xsl:variable name="text">
    <p>Document contains <xsl:value-of select="$how-many-elements"/>
        elements.</p>
</xsl:variable>

<xsl:template match="/">
    ... <xsl:value-of select="$text"/> ...
</xsl:template>
```
Template parameters

- `param in template`
- `with-param in apply-templates`

Example

```xml
<xsl:template match="document">
  <xsl:apply-templates select="chapter">
    <xsl:with-param name="prefix" select="'Section '"></xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="chapter">
  <xsl:param name="prefix" select="'Chapter ' />
  <li><xsl:value-of select="$prefix"/></li>
  <xsl:apply-templates/>
</xsl:template>
```
Named templates

- Attribute `name` in template
- Invocation with `call-template`
- No change of current node (unlike `apply-templates`).
- Recursion possible

**Example**

```xml
<xsl:template name="describe-element">
  <p>Element with name <xsl:value-of select="local-name()"/></p>
</xsl:template>

<xsl:template match="/">
  <html><body>
    <h1>All elements:</h1>
    <xsl:for-each select="//*">
      <xsl:call-template name="describe-element"/>
    </xsl:for-each>
  </body></html>
</xsl:template>
```
Parameters and recursion in named templates

“Programming” in XSLT

Factorial (accumulator solution)

```xml
<xsl:template name="fac">
  <xsl:param name="n"/>
  <xsl:param name="res" select="1"/>
  <xsl:choose>
    <xsl:when test="$n &gt; 1">
      <xsl:call-template name="fac">
        <xsl:with-param name="n" select="$n - 1"/>
        <xsl:with-param name="res" select="$n * $res"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$res"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```
Custom function definitions (XSLT 2.0)

Factorial (naive solution)

```xml
<xsl:function name="loc:fac">
  <xsl:param name="n"/>
  <xsl:sequence select="if($n &lt;= 1)
                then 1
                else $n * loc:fac($n - 1)"/>
</xsl:function>
```
Result tree serialisation

- Result of transformation — XPath tree
- Serialization — writing result as sequence of bytes
- Serialization methods:
  - xml,
  - html,
  - xhtml (only XSLT 2.0),
  - text.

Specification in stylesheet

```xml
<xsl:output method="html" encoding="iso-8859-2"/>
```
**XSLT Features**  
**Serialisation methods**

## Additional serialization parameters

- **encoding** — character encoding
- **version** — XML or HTML version
- **doctype-public, doctype-system** — DOCTYPE declaration
- **indent** — automatic indentation

### Specification in stylesheet

```xml
<xsl:output method="xhtml" version="1.0" encoding="utf-8"
    doctype-public="-//W3C//DTD XHTML 1.1//EN"
    doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

### Beginning of result document

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>...
```
Additional serialization parameters

- **encoding** — character encoding
- **version** — XML or HTML version
- **doctype-public, doctype-system** — DOCTYPE declaration
- **indent** — automatic indentation

**Specification in stylesheet**

```xml
<xsl:output method="xhtml" version="1.0" encoding="utf-8"
    doctype-public="-//W3C//DTD XHTML 1.1//EN"
    doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

**Beginning of result document**

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>...
```
Features of XSLT 2.0 not present in XSLT 1.0

- XPath 2.0 with sequences, \texttt{if}, XML Schema types
- Function definitions
- Grouping and \texttt{perform-sort}
- Additional result trees (and output files)
- Regex-based text processing (\texttt{analyze-string})
- ...
Temporary result tree fragments

- **XSLT 1.0** — distinct types `node-set` and `result-tree-fragment`
  - not allowed to mix
  - not allowed to process result tree fragment again
- **XSLT 2.0** — no such restriction

**XSLT 2.0 but not XSLT 1.0**

```xml
<xsl:variable name="tmp">
  <xsl:apply-templates select="document"/>
</xsl:variable>

<xsl:apply-templates select="$tmp" mode="correct"/>
```
Things that didn’t fit in

- Numbering
- Diagnostic messages, debugging, fallback
- Regex-based text processing (XSLT 2.0).
- Schema-aware processing (XSLT 2.0).
- Serialisation and...:
  - namespaces
  - CDATA sections
  - character maps
XSLT applications

**Typical applications**

- Text document presentation (HTML, XSL-FO)
- “Database” documents presentation
- Conversion between different XML formats
- Filters, reports
- ...

**Sophisticated applications**

- Verification of integrity constraints not expressible in XML Schema
- XSLT as XSLT result
- Generating scripts and configuration files
- ...

Patryk Czarnik