XSLT – Transforming XML Documents

Pierre Genevès
CNRS

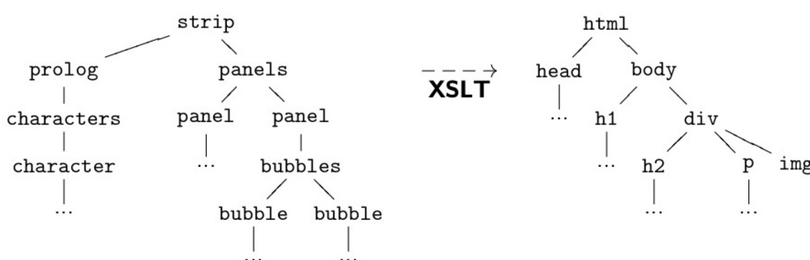
(some slides based on Marc H. Scholl's ones)

University of Grenoble, 2011–2012

XSLT

- XSLT is a *declarative* and ***functional*** language...
- ... for transforming XML documents into other XML documents
- Uses the XML syntax
- Uses the XPath language
- A W3C recommendation (www.w3.org/TR/xslt)
- Allows transforming documents of a given type into another
- Often used for presentation purposes (typical example: XML $\rightarrow$ XHTML)

Example: XML → XHTML transformation via XSLT:



Separating Content from Presentation

Contrary to when style information is hard-coded into the content, **separation of style from content** allows for the same data to be presented in many ways:

1. **Reuse** fragments of data
   (same contents looks different depending on context),
2. **multiple output formats**
   (media [online, paper], sizes, devices [workstation, handheld]),
3. styles tailored to **reader's preference**
   (accessibility issues, audio rendering),
4. **standardized** styles
   (corporate identity, web site identity),
5. **freedom from style**
   (do not bother tech writers with layout issues).

XSLT Stylesheets

- An **XSL stylesheet** defines a set of **templates** ("tree patterns and actions").

  Each template . . .

  1. **matches** specific elements in the XML doc tree, and then
  2. **constructs** the contribution that the elements make to the transformed tree.

- XSL is **an application of XML** itself:
  - Each XSL stylesheet is an XML document,
  - elements with a **name prefix**[34] `xsl:` are part of the XSLT language,
  - non-"`xsl:`" elements are used to construct the transformed tree.

---

[34] More correctly: elements in the **namespace** `http://www.w3.org/1999/XSL/Transform`. For details on namespaces, see `http://www.w3.org/TR/REC-xml-names`.

Example: Transform text markup into HTML style paragraph and emphasis tags:

style.xsl

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3
4  <xsl:template match="para">
5    <p><xsl:apply-templates/></p>
6  </xsl:template>
7
8  <xsl:template match="emphasis">
9    <i><xsl:apply-templates/></i>
10 </xsl:template>
11
12 </xsl:stylesheet>
```

input.xml

```
1  <?xml version="1.0"?>
2  <para>This is a <emphasis>test</emphasis>.</para>
```

output.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <p>This is a <i>test</i>.</p>
```

N.B. Note how XSLT acts like a **tree transformer** in this simple example.

XSLT Templates

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

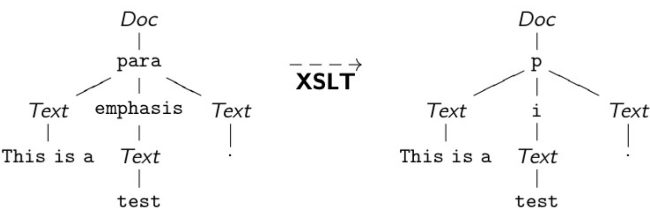
- *e* is an **XPath expression**,
  selecting the nodes in the document tree XSLT will apply the template to,
- *cons* is the **result constructor**,
  describing the transformation result that the XSLT processor will produce for the nodes selected by *e*.

  N.B. "`xsl:`" elements in *cons* will be interpreted by the XSLT processor.

  `<xsl:apply-templates/>` applies the template matching process **recursively to all child nodes** of the matched node.

Applying the Template...

The actual tree transformation in our previous example goes like this:



Something else must be going on here:

1. The *Text* nodes have *automatically* been copied into the result tree.
2. How could the `para` and `emphasis` elements match anyway? (The XPath patterns for both templates used *relative* paths expressions.)

Default Templates

Each XSLT stylesheet contains two **default templates** which

1. **copy Text and Attr (attribute) nodes** into the result tree:

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

2. **recursively drive the matching process**, starting from the document root:

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

`<xsl:value-of select="e"/>` copies those nodes into the result tree that are reachable by the XPath expression *e* (context node is the matched node).
The default templates may be **overridden**.

Overriding default XSLT templates

What would be the effect of applying the following XSLT stylesheet?

style.xsl

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
3
4  <xsl:template match="text()">foo</xsl:template>
5
6  <xsl:template match="para">
7    <p><xsl:apply-templates/></p>
8  </xsl:template>
9
10 <xsl:template match="emphasis">
11   <i><xsl:apply-templates/></i>
12 </xsl:template>
13
14 </xsl:stylesheet>
```

More XSLT defaults

XSLT contains the following additional default template. Explain its effect.

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

Intermediate Summary

XSLT Instruction	**Effect**
`<xsl:template match="e">` *cons* `</xsl:template>`	Replace nodes matching path expression *e* by *cons*.
`<xsl:apply-templates select="e"/>`	Initiate template matching for those nodes returned by path expression *e* (default: path $e =$ `child::node()`).
`<xsl:value-of select="e"/>`	Returns the (*string value*[35] of the) result of XPath expression *e*.

[35] Read: The *string value* of an XML element node is the concatenation of the contents of all *Text* nodes in the subtree below that element (in document order).

Use of XPath in XSLT

XPath is used with two **very distinct** semantics:

1. for selecting, computing, and producing (the usual XPath semantics):
   - selecting the next tree nodes that will be treated
   - computing boolean conditions
   - producing text in the result tree

→ used in XSLT instructions: `apply-templates`, `value-of`, `if`, ...

2. for specifying template *patterns*

→ used in XSLT instruction: `template`

Semantics of *patterns*: matching

```
<xsl:template match="pattern">
     <!-- result fragments and instructions -->
</xsl:template>
```

A *pattern* identifies nodes to which the template applies

A *pattern* is a restricted XPath expression:

- A union of *location paths*, where:
- each *location path* contains *location steps* separated by // and /
- only the axes `child` and `attribute` are permitted in the *location steps*

A *pattern* *matches* a node *n* if

- starting from *some* node in the tree:
- node *n* is *included* in the resulting *node-set*

Example: `appendix//ulist/item`

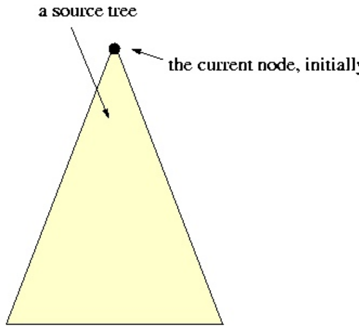
Summary: Recursive Transformation Mechanism

→ **Start:** the result of a transformation is obtained by processing the root node of the source document

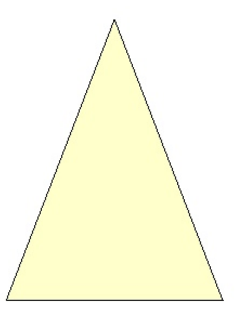
- A node is processed as follows:
  1. the most appropriate *template* is found (the one with the most specific *pattern*)
  2. this *template* is instanciated:
     - the result fragment is created
     - processing continues recursively (owing to the instructions of the *template* body that select other nodes)
- A node list is processed by processing each node in order and concatenating the results
- Source nodes are processed only if they have been selected by some instruction

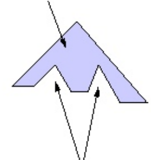
→ **Termination:** recursion stops when no more source node is selected

Recursive Transformation Mechanism

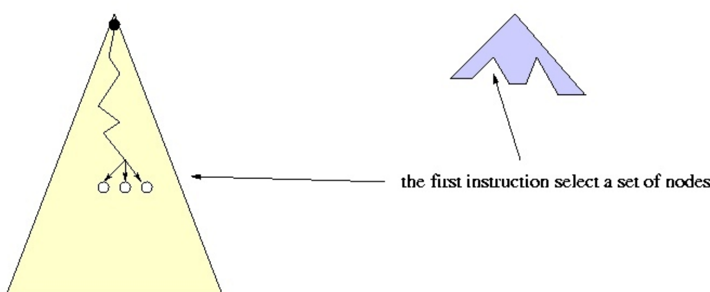


Recursive Transformation Mechanism

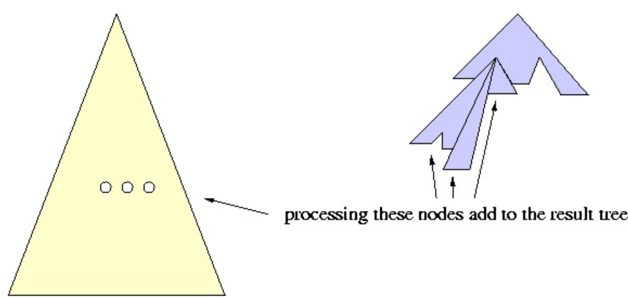




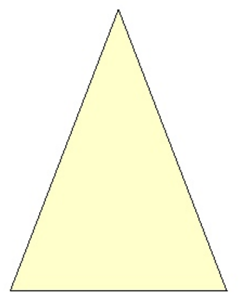
Recursive Transformation Mechanism

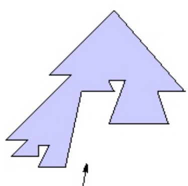


Recursive Transformation Mechanism



Recursive Transformation Mechanism





when there are no more instructions
to process, the result tree is done

Questions

✎ Recursion in XSLT

Explain the effect of the following XSLT stylesheet

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

✎ What would be the effect of applying an empty stylesheet?

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

Example: Dilbert Comic Strips...

Transform a DilbertML document into an HTML representation that reflects the comic strip's story:

- From the `prolog`, generate the HTML header, title, heading, copyright information.
- From `characters`, generate an unordered HTML list (`ul`) of all featured comic characters.
- For all `panels`, reproduce the `scene` as well as all spoken `bubbles`, indicating who is speaking to whom (if available).

Note:
`<xsl:if test="`*p*`"/>` *cons* `</xsl:if>` reproduces *cons* in the result tree, if the XPath predicate *p* evaluates to true.[36]

[36] Remember from XPath: an empty node sequence is interpreted as false, a non-empty sequence as true.

dilbert.xsl

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0"
3       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4
5  <!-- Generate document head and body, insert prolog information -->
6  <xsl:template match="/">
7     <html>
8       <head>
9         <title>
10          <xsl:value-of select="/strip/prolog/series"/>
11        </title>
12      </head>
13      <body>
14        <h1> <xsl:value-of select="/strip/prolog/series"/>
15        </h1>
16        <p>A comic series by
17           <xsl:value-of select="/strip/prolog/author"/>,
18           copyright (C)
19           <xsl:value-of select="/strip/@year"/> by
20           <xsl:value-of select="/strip/@copyright"/>
21        </p>
22        <xsl:apply-templates/>
23      </body>
```

```
24     </html>
25   </xsl:template>
26
27   <!-- The next 2 templates generate the
28        "Featured Characters" bullet list -->
29   <xsl:template match="characters">
30      <h2> Featured Characters </h2>
31      <ul>
32        <xsl:apply-templates/>
33      </ul>
34   </xsl:template>
35
36   <xsl:template match="character">
37      <li> <xsl:value-of select="."/> </li>
38   </xsl:template>
39
40   <!-- Reproduce the panel and the scene it displays -->
41   <xsl:template match="panel">
42     <h3> Panel <xsl:value-of select="@no"/> </h3>
43     <p> <xsl:value-of select="scene"/> </p>
44     <xsl:apply-templates select="bubbles"/>
45   </xsl:template>
46
```

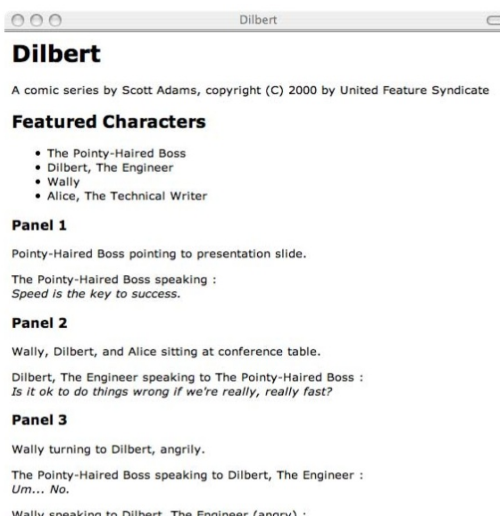
```
47  <!-- Reproduce spoken text, indicating tone and
48       who is speaking to whom -->
49  <xsl:template match="bubble">
50    <p> <xsl:value-of select="id(@speaker)"/> speaking
51        <xsl:if test="@to">
52          to <xsl:value-of select="id(@to)"/>
53        </xsl:if>
54        <xsl:if test="@tone">
55          (<xsl:value-of select="@tone"/>)
56        </xsl:if>
57        :<br/>
58        <em>
59           <xsl:value-of select="."/>
60        </em>
61    </p>
62  </xsl:template>
63
64  <!-- Suppress all other text/attributes -->
65  <xsl:template match="text()|@*"/>
66
67  </xsl:stylesheet>
```

dilbert.html

```
<html>
<head> <title>Dilbert</title> </head>
<body>
  <h1>Dilbert</h1>
  <p>A comic series by Scott Adams, copyright
    (C) 2000 by United Feature Syndicate </p>
  <h2> Featured Characters </h2>
  <ul>
    <li>The Pointy-Haired Boss</li>
    <li>Dilbert, The Engineer</li>
    <li>Wally</li>
    <li>Alice, The Technical Writer</li>
  </ul>
  <h3> Panel 1</h3>
  <p>Pointy-Haired Boss pointing to presentation slide.
  </p>
  <p>The Pointy-Haired Boss speaking : <br>
    <em>Speed is the key to success.</em>
  </p>
  <h3> Panel 2</h3>
  <p>Wally, Dilbert, and Alice sitting at conference table.
  </p>
  <p>Dilbert, The Engineer speaking
```

```
24        to The Pointy-Haired Boss : <br>
25       <em>Is it ok to do things wrong if
26           we're really, really fast?</em>
27     </p>
28     <h3> Panel 3</h3>
29     <p>Wally turning to Dilbert, angrily.
30     </p>
31     <p>The Pointy-Haired Boss speaking
32        to Dilbert, The Engineer : <br>
33       <em>Um... No.</em>
34     </p>
35     <p>Wally speaking
36        to Dilbert, The Engineer (angry) : <br>
37       <em>Now I'm all confused.  Thank you very much.</em>
38     </p>
39   </body>
40   </html>
```

Screenshot of Firefox Rendering file dilbert.html



Conflict Resolution 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 *e*	**Priority**
*	−0.5
ns:*	−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:
  `<xsl:template priority="p" ...>`.

Modes

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
`xsl:choose`, `xsl:when`	switch statement (a la C)
`xsl::call-template`	explicitly invoke a (named) template
`xsl::for-each`	replicate result construction for a sequence of nodes
`xsl::if`	conditional statement (XPath test)
`xsl::element`	generate element (name) dynamically
`xsl::copy`	copy the current node
`xsl::import`	import instructions from another stylesheet
`xsl::output`	influence XSLT processor's output behavior
`xsl::variable`	set/read variables

Question

What is the effect of the following?

?????.xsl

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

Zoom on Variables

- A variable x is bound to a value (declared) with:

```
<xsl:variable name="x" select="p"/>
```

  where p is an XPath expression that sets the value.

- x can then referred to (read) with $x, **in scope** of the variable definition

- **Scope**: a variable is global if it's declared as a top-level element, and local if it's declared within a template.

- There is no way to change the value of a defined variable.

→ XSLT is a **functional** language (no side-effect)

Concluding Remarks

- XSLT is convenient for defining simple/complex (source-driven or target-driven) transformations
- It is **powerful** (Turing-complete)

  $\rightarrow$ in general: termination, inversion, ... are undecidable

- **Many implementations** (including open source)
- For more details: W3C http://www.w3.org/TR/xslt
- XSLT has however some limitations:
  - Incremental transformation (theoretically possible since the language is functional, but not available in most XSLT processors)
  - Streaming XML transformations (currently under active research)
  - Static type-checking (currently under active research)

The XSLT Type-Checking Problem

Given a type T_{in}, an XSLT stylesheet f and a type T_{out}, does $f(t) \in T_{\text{out}}$ for all $t \in T_{\text{in}}$?