XSLT Notes *

Jef Wijsen

May 10, 2019

Contents

*The larger part of this document is based on the W3C specifications and recommendations available at `http://www.w3.org/`. The examples are mine.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="tree-view.xsl"?>
<catalog tax="no" city="Mons" xmlns="iURL" xmlns:q="qURL">
  <!--A comment-->
  Spaces   are
  <car EUR="12000">Peugeot Partner</car>
  not easy
  <q:bike cm="56"/>
</catalog>
```

Figure 1: Well-formed XML document.

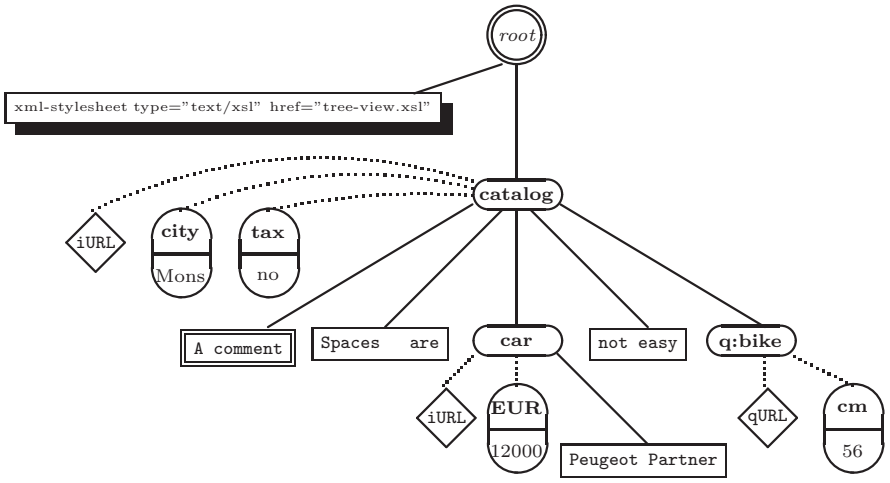


Figure 2: Tree representation of the well-formed XML document in Fig. 1.

1 XML

1.1 Well-Formed XML Document

Fig. 1 shows a *well-formed* XML document. *Elements* of *type* `car` are delimited by the *start-tag* `<car>` and the *end-tag* `</car>`. The text between the start-tag and end-tag is called the element's *content*. The element of type `bike` has no content and is said to be *empty*. The element of type `catalog` contains five *children*. An element is said to have *mixed content* if it contains character data, optionally (sic) interspersed with child elements. *Attributes* are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags. An attribute name *must not* appear more than once in the same start-tag or empty-element tag. A well-formed XML document has a single *root element*, also called the *document element*. The root element in the running example is of type `catalog`.

1.2 XML Trees

XPath models an XML document as a tree. The tree contains nodes. There are seven types of nodes:

1. element nodes There is an element node for every element in the document.

2. the root node Note that the element node for the document element (`catalog` in the

current example) is *not* the root node, but a child of the root node. So the root node has no corresponding element or attribute in the XML document.

3. **text nodes** Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node.

4. **attribute nodes** Each element node has an associated (possibly empty) set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is *not* a child of its parent element.[1]

5. **namespace nodes** As for attribute nodes, a namespace node is not a child of its parent element.

6. **processing instruction nodes**

7. **comment nodes**

Fig. 2 shows the tree representation of the XML document in Fig. 1, containing sixteen nodes: the root node labeled *root*, three element nodes, three text nodes, four attribute nodes, three namespace nodes (diamonds), one processing instruction node (shadowbox), and one comment node. Note that the XML declaration `<?xml version="1.0"?>` is not part of the tree. Attribute order is not significant in XML.

Hint 1 The XSLT and CSS stylesheets `tree-view.xsl` and `tree-view.css` available at `http://skew.org/xml/stylesheets/treeview/html/` transform any XML document into an HTML document showing the tree representation of the XML document. Simply download both stylesheets and add to your XML document a processing instruction

```
<?xml-stylesheet type="text/xsl" href="tree-view.xsl"?>
```

to ask Microsoft Internet Explorer to do the transformation.

1.3 Valid XML Document

A *valid* XML document is a well-formed XML document which also conforms to the rules of a *Document Type Definition* (DTD). W3C supports an alternative to DTD called XML Schema.

The DTD of Fig. 3 specifies that a catalog contains zero or more cars, bikes, or go-karts. All vehicles have a price expressed in EUR or BEF. In addition, bikes store the frame height. Cars also contain the model and optionally the color. Fig. 4 shows a valid XML document; the tree representation appears in Fig. 5.

2 XPath

2.1 Location Step

A *location step* selects nodes relative to the *context node*; it has three parts:

1. an *axis*,
2. a *node test* preceded by `::`, and
3. zero or more *predicates* each in square brackets, which use arbitrary expressions to further refine the set of nodes selected by the location step.

[1] For this reason, I use a dotted line between an attribute node and its parent element node.

```
<!ELEMENT catalog (car|bike|go-kart)*>
<!ELEMENT car (model,color?,price)>
<!ELEMENT bike (height,price)>
<!ELEMENT go-kart (price)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT color (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT height (#PCDATA)>
<!ATTLIST price unit (EUR|BEF) #REQUIRED>
```

Figure 3: DTD stored at `http://ssi.umh.ac.be/jefXML/cat.dtd`.

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM "http://ssi.umh.ac.be/jefXML/cat.dtd">
<catalog>
   <car>
      <model>Renault CLIO</model>
      <color>blue</color>
      <price unit="BEF">115000</price>
   </car>
   <bike>
      <height>56</height>
      <price unit="EUR">500</price>
   </bike>
   <go-kart>
      <price unit="BEF">3000</price>
   </go-kart>
   <car>
      <model>Peugeot Partner</model>
      <color>red</color>
      <price unit="EUR">12000</price>
   </car>
</catalog>
```

Figure 4: Valid XML document.

For example,

```
child::price[attribute::unit="EUR"]
```

selects the `price` element children of the context node that have a `unit` attribute with value `EUR`. The node test `price` is true for a `price` element node.

2.2 Axes

An *axis* specifies the tree relationship between the "target" nodes and the context node. The thirteen axes are:

- `self`
  The `self` axis contains just the context node itself.
- `parent` and `child`
  The `parent` axis contains the parent of the context node. Every node other than the root node has exactly one parent, which is either an element node or the root node.

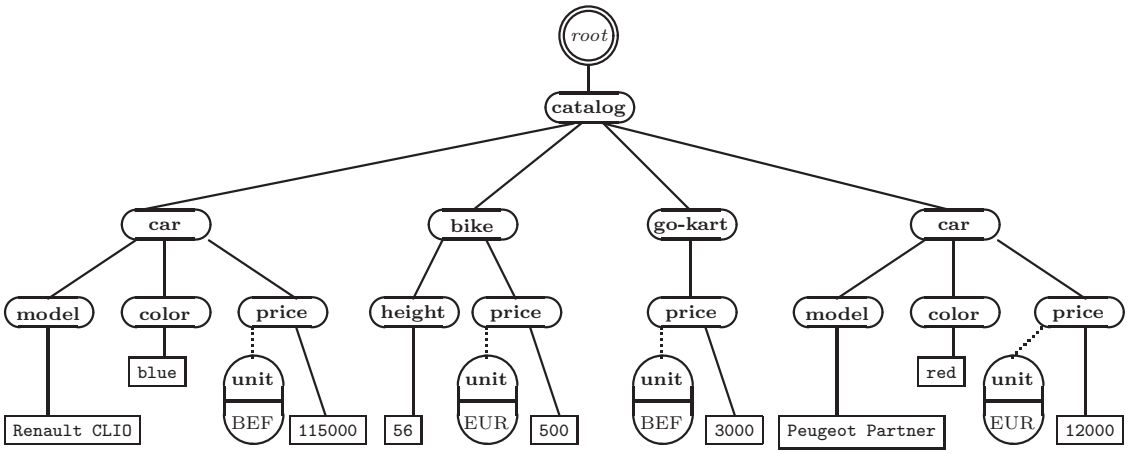


Figure 5: Tree representation of the valid XML document in Fig. 4.

The `child` axis contains the children of the context node. Since an attribute is *not* a child of its parent element, this axis will contain no attribute nodes. The attributes of an element context node are in the `attribute` axis:

- `attribute`
  The `attribute` axis contains the attributes of the context node; the axis will be empty unless the context node is an element.
- `ancestor` and `descendant`
  The `ancestor` axis will always include the root node, unless the context node is the root node.

  The `descendant` axis never contains attribute or namespace nodes.
- `ancestor-or-self` and `descendant-or-self`
- `preceding` and `following`
  The `preceding` axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and *excluding attribute nodes* and namespace nodes.

  The `following` axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and *excluding attribute nodes* and namespace nodes.
- `preceding-sibling` and `following-sibling`
  Two nodes are siblings if they have the same parent node. The `preceding-sibling` axis contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty.
- `namespace`
  The `namespace` axis contains the namespace nodes of the context node; the axis will be empty unless the context node is an element.

The `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition a document (ignoring `attribute` and `namespace` nodes): they do not overlap and together they contain all the nodes in the document.

Hint 2 Although attribute order is not significant in XML, this does not mean that the mutual order of an attribute node and an element node with the same parent is not significant. For example, consider the XML document:

```
<?xml version="1.0"?>
<a c="0">
   <b/>
</a>
```

Assume that the `c` attribute node is the context node. Then the `following` axis contains the `b` element, and the `preceding` axis is empty. However, I found that certain XSLT processors behave differently, leaving both axes empty.

2.3 Node Tests

2.3.1 Tests for Any Node

A node test `node()` is true for any node of any type whatsoever. For example,

- `child::node()` selects all the children of the context node, *whatever their node type*. It will *not* select attribute nodes, as an attribute is *not* a child of its parent element.

2.3.2 Tests for Element, Attribute, and Namespace Nodes

A node test `mylabel` is true for `mylabel` element nodes and for `mylabel` attribute nodes. For example,

- `child::bike` will select all `bike` element children of the context node.
- `attribute::unit` selects the `unit` attribute of the context node (there can be at most one).

A node test `*` is true for any node of the *principal node type*. For the attribute axis, the principal node type is attribute. For the namespace axis, the principal node type is namespace. For other axes, the principal node type is element. For example,

- `child::*` will select all element children of the context node, and
- `attribute::*` will select all attributes of the context node.

Namespace nodes are rarely handled explicitly.

2.3.3 Tests for Text, Comment, and Processing-Instruction Nodes

The node test `text()` is true for any text node. Similarly, the node test `comment()` is true for any comment node, and the node test `processing-instruction()` is true for any processing instruction. For example,

- `child::text()` will select the text node children of the context node.
- `child::comment()` will select the comment node children of the context node.

Note the different treatment of attribute and namespace nodes on the one hand (they have their own axes, but no separate node tests), and text, processing instruction and comment nodes on the other hand (separate node tests, but no axes).

2.4 Location Path

A *relative location path* consists of a sequence of one or more *location steps* separated by /, and selects a set of nodes relative to the context node. An *absolute location path* consists of / optionally followed by a relative location path, and selects a set of nodes relative to the root node of the document. For example,

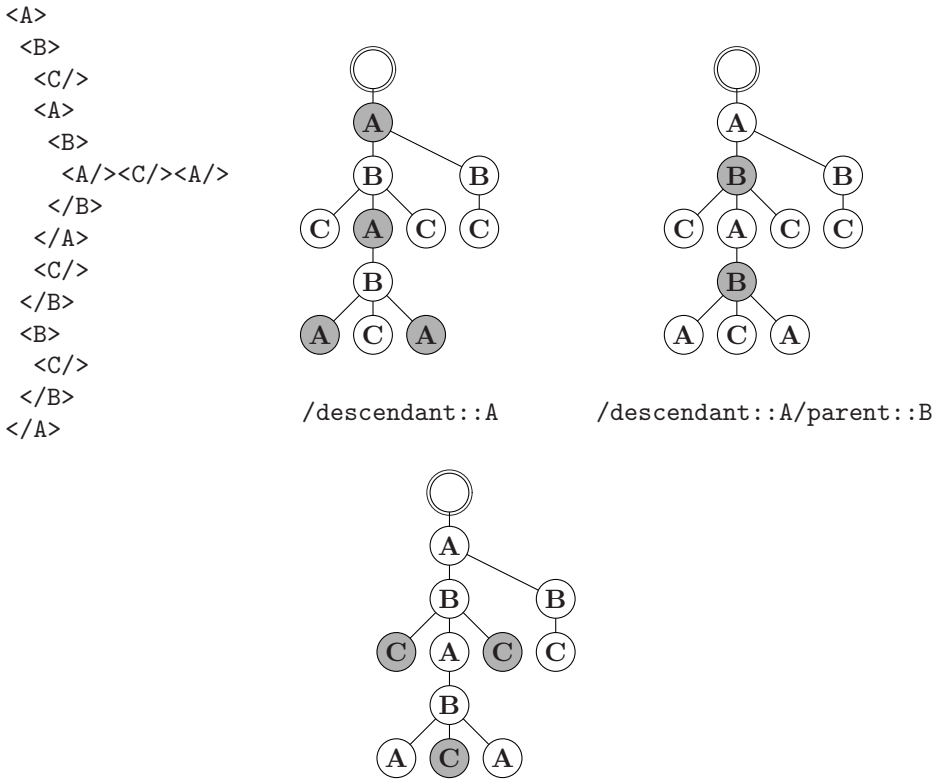


Figure 6: Selected element nodes are grey coloured.

```
child::*/child::color
```

selects all `color` grandchildren of the context node. A more involved example is shown in Figure 6.

A location path can be used as an expression. The expression returns the set of nodes selected by the path. The `|` operator computes the union of its operands, which must be node-sets.

2.5 Abbreviations

The most important abbreviation is that `child::` can be omitted from a location step. In effect, `child` is the default axis. Abbreviations are listed in the following table:

Shorthand	Unabbreviated syntax
can be omitted	`child::`
`//`	`/descendant-or-self::node()/`
`@`	`attribute::`
`.`	`self::node()`
`..`	`parent::node()`

For example,

```
.//price
```

will select all `price` descendant elements of the context node. It is an abbreviation for:

```
self::node()/descendant-or-self::node()/child::price
```

2.6 Predicates

Predicates are actually XPath *expressions*. XPath expressions can be of four types: node-set, boolean, number, or string. A node-set expression is really the same thing as a location path. If a node-set expression is used as a predicate, then the (implicit) conversion to true/false is natural: an empty node-set is converted into false, a nonempty node-set into true. For example,

- `child::car[descendant::color]` selects `car` element children of the context node that have a `color` element descendant.

In general, a node-set expression returns a *sequence* of nodes: all nodes matching the expression, ordered according to document order. Node-sets can be evaluated as sequences of strings or numbers that can be manipulated by the XPath operators:

```
and or = != < <= > >= + - * div mod
```

and functions including:

```
count sum position() last() starts-with
```

For example,

- `car[color='blue']` selects the `car` element children of the context node that contain a `color` element child whose value is blue.
- `car[count(color)>1]` selects the `car` element children of the context node that contain more than one `color` element child.
- `car[not(starts-with(model,'R'))]` selects the `car` element children of the context node that contain a `model` whose value does not begin with the letter R.
- `descendant::car[position()=2]` selects the second (in document order) `car` element descendant of the context node. This can be abbreviated as `descendant::car[2]`.
- `descendant::car[position()=last()-1]` selects the last but one (in document order) `car` element descendant of the context node.
- ```
  descendant::car[(price*40.3399>500000 and price/@unit='EUR') or
                  (price>500000 and price/@unit='BEF')]
  ```
  selects all `car` element descendants of the context node whose price expressed in BEF is greater than 500000.
- Predicates can be nested, i.e. square brackets can appear within square brackets. The preceding location step can also be written:

  ```
  descendant::car[(price[@unit='BEF']>500000) or
                  (40.3399*price[@unit='EUR']>500000)]
  ```
- `catalog[sum(*/price)>999999]` selects all `catalog` element children of the context node with the total price of all vehicles exceeding 999999.

2.7 Comparison of Node-sets

If an operand of `=`, `!=`, `<=`, `<`, `>=` or `>` is a node-set, then "existential" semantics apply. For example, the following XPath expression returns the entire catalog if there *exists* a blue product; otherwise the result is empty.
  ```

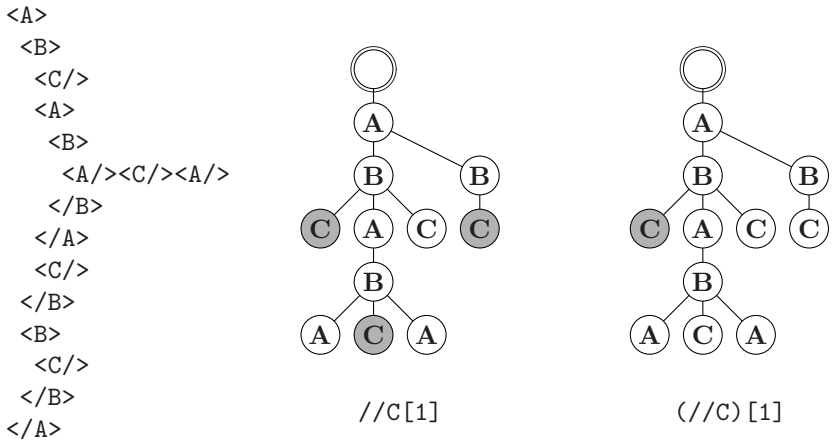


Figure 7: Selected element nodes are grey coloured.

```
/catalog[descendant::color='blue']
```

Consequently, the following XPath expressions are *not* equivalent (why not?):

`/catalog[descendant::color!='blue']` $\quad\leadsto\quad \exists c(c \neq \text{blue})$

`/catalog[not(descendant::color='blue')]` $\quad\leadsto\quad \neg\exists c(c = \text{blue}) \equiv \forall c(c \neq \text{blue})$

2.8 Pitfalls

Thanks to Jan Paredaens. Figure 7 shows that `//C[1]` can return more than one `C` element node. Can you explain why? By using parentheses, you can select the first `C` element node in the document: `(//C)[1]`.

Figure 8 shows that equality seems not transitive. Can you explain why no element node is selected in the third tree?

3 XSLT

3.1 XSL Stylesheet

A *template rule* has the form:

```
<xsl:template match="the match pattern">
   the template
</xsl:template>
```

An XSL *stylesheet* is a family of such template rules.

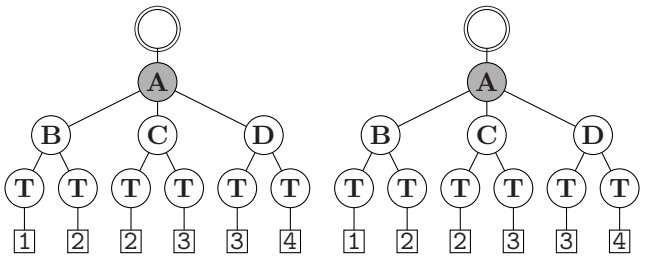
3.2 XSLT Processing Model

The XSLT processor takes as its input an XML document and a stylesheet, and produces an output XML document:

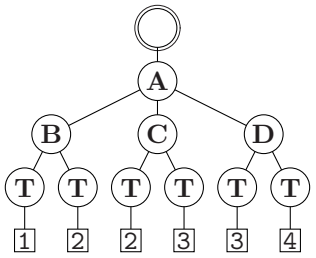
$$\text{XML document} + \text{XSL stylesheet} \xrightarrow{\text{xslt}} \text{result tree}$$

The xslt program can be understood as follows:

```
<A>
 <B>
   <T>1</T>
   <T>2</T>
 </B>
 <C>
   <T>2</T>
   <T>3</T>
 </C>
 <D>
   <T>3</T>
   <T>4</T>
 </D>
</A>
```



/A[B/T=C/T] /A[C/T=D/T]



/A[B/T=D/T]

Figure 8: Selected nodes are grey coloured. Equality seems not transitive.

```
program xslt
    procedure process(aNode)
    begin
    find the template rule with pattern that best matches aNode (see Section 3.3)
    execute the template with aNode as the current node (see Section 3.4)
    end
begin
process(theRootNode)
end
```

3.3 Matching

The *match pattern* is a restricted location path (see Section 2.4). The restrictions include:

- In any location step, the axis must be either `child` or `attribute`. Recall that a location step is of the form *axis*`::`*node-test*[*predicate*]*.
- In a location path, location steps can be separated not only by `/` but also by `//`. When `//` is used in this way, it is not regarded as a shorthand.

A pattern p is defined to *match* a node n if and only if there is a possible context node c such that when the pattern p is evaluated relative to c, the node n is a member of the resulting node-set.

For example, the pattern `bike` matches every `bike` element node. Recall that `bike` is a shorthand for `child::bike`. Let n be a `bike` element node, and let c be the parent of n (c is either another element node or the root node). Evaluating `bike` relative to c results in a node-set containing n. It is correct to conclude that n matches `bike`.

Hint 3 You may find it easier to understand the meaning of a pattern by evaluating it from right to left. For example, `catalog//bike` may be read as "any `bike` element that has a `catalog` element ancestor."

Similarly, the pattern `node()` matches any node other than an attribute node, a namespace node, and the root node. Recall that `node()` is a shorthand for `child::node()`. Since an attribute node is *not* a child of its parent element, attribute nodes will not be matched. Since the root node is *not* a child of any other node, the root node will not be matched. Always using the same reasoning, we can obtain the following table:

	match=						
	`"node()"`	`"*"`	`"@*"`	`"/"`	`"text()"`	`"comment()"`	`"processing -instruction()"`
root	−	−	−	+	−	−	−
element	+	+	−	−	−	−	−
text	+	−	−	−	+	−	−
comment	+	−	−	−	−	+	−
processing -instruction	+	−	−	−	−	−	+
attribute	−	−	+	−	−	−	−
namespace	−	−	−	−	−	−	−

There is no pattern that can match a namespace node.

3.4 Executing the Template

Typically the template contains one or more of the following elements:

- `<xsl:apply-templates select="`*node-set*`"/>`
  The *node-set* is given by a path that selects nodes relative to the current node (i.e. the context node comes from the current node). The execution of this statement can be understood as follows:

  $$\text{for each node } n \text{ in } \textit{node-set}, \ \mathsf{process}(n) \ ,$$

  where process() as described in Section 3.2. Typically, `xsl:apply-templates` is used to process only nodes that are descendants of the current node; this ensures termination of the recursion. The common use of `xsl:apply-templates` is

  ```
  <xsl:apply-templates select="child::node()"/> ,
  ```

  i.e. continue processing the children of the current node; it can be shortened into:

  ```
  <xsl:apply-templates/> .
  ```

  Multiple `xsl:apply-templates` elements can be used within a single template to do simple reordering.

- `<xsl:copy-of select="`*node-set*`"/>`
  Copy the *node-set* over to the result tree. Copying an element node copies the attribute nodes, namespace nodes and children of the element node as well as the element node itself.

- `<xsl:value-of select="`*node-set*`"/>`
  Create a text node in the result tree. The text node is obtained by converting *node-set* into a string. The string-value of an element node is the concatenation, in document order, of all text node descendants.

3.5 Built-In Template Rules

Take the XML document of Fig. 4 and process it with an empty stylesheet, i.e. a stylesheet containing the empty element tag:

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

The result contains all text nodes in document order:

```
Renault CLIOblue115000565003000Peugeot Partnerred12000
```

The explanation for this output is that XSLT contains some built-in template rules. The most important built-in rule applies to both element nodes and the root node, and can be expressed like this:

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

`/` matches the root node, and `*` matches any element node (see Section 3.3). So `/ | *` is XPath shorthand for "any element node or the root node." So this rule is simply there to make sure that every element, from the root on down, is processed with `<xsl:apply-templates/>` if you don't supply some other rule. If you do supply another rule, it overrides the corresponding built-in rule. There is also a built-in template rule for text and attribute nodes that copies text through:

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

Note incidentally the absence of EUR or BEF in the output obtained with the empty stylesheet.[2] The built-in template rule for processing instructions and comments is to do nothing:

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

The built-in template rule for namespace nodes is also to do nothing. There is no pattern that can match a namespace node; so, the built-in template rule is the only template rule that is applied for namespace nodes.

3.6 Priority Processing

The aim of the following (deficient) stylesheet was to ignore cheap cars, to show medium-prized cars in green, and expensive ones in red.

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

   <xsl:template match="car[price>99999]">
      <FONT COLOR="red"><xsl:value-of select="model"/></FONT>
   </xsl:template>

   <xsl:template match="car[price>10000]">
      <FONT COLOR="green"><xsl:value-of select="model"/></FONT>
   </xsl:template>

   <xsl:template match="car"/>
</xsl:stylesheet>
```

The stylesheet contains three template rules that could match a `car` element node. The third rule is less specific than the two preceding rules, which are equally specific.[3] The basic assumption is that rules that are more specific take precedence over rules that are more general. Therefore, cars with a price > 10000 and ≤ 99999 must be handled by the second rule (they will appear in green), not by the last one. For cars with a price > 99999, which match equally well the first and the second rule, the following passage in the W3C Recommendation (`http://www.w3.org/TR/xslt`) applies:

> *"It is an error if this leaves more than one matching template rule. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the matching template rules that are left, the one that occurs last in the stylesheet."*

Actually, Microsoft Internet Explorer does not signal "the error" and chooses among the equally specific rules the one that occurs last, which is conform to the recommendation. Under this choice, the first rule could be made executable by placing it after the second one in the stylesheet. However, it is more reliable to give the first rule a higher priority by using the XSLT `priority` attribute.

[2] The default built-in processing starts with the root node and then recursively traverses the `child` axis; so it will never reach an attribute node.

[3] XSLT is not aware, for obvious reasons, that prices greater than 99999 are necessarily greater than 10000.

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

   <xsl:template match="car[price>99999]" priority="2">
      <FONT COLOR="red"><xsl:value-of select="model"/></FONT>
   </xsl:template>

   <xsl:template match="car[price>10000]" priority="1">
      <FONT COLOR="green"><xsl:value-of select="model"/></FONT>
   </xsl:template>

   <xsl:template match="car"/>
</xsl:stylesheet>
```

Now cars with a price over 99999 will be handled by the first rule (they appear in red).

3.7 Multiple Processing

You can process the same node multiple times by using the XSLT `mode` attribute.

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="/">
      <HTML>
      <xsl:apply-templates select="//car" mode="theblues"/>
      <xsl:apply-templates select="//car" mode="blackout"/>
      </HTML>
   </xsl:template>

   <xsl:template match="car" mode="theblues">
      <FONT COLOR="blue"><xsl:value-of select="."/></FONT>
   </xsl:template>

   <xsl:template match="car" mode="blackout">
      <FONT COLOR="black"><xsl:value-of select="."/></FONT>
   </xsl:template>
</xsl:stylesheet>
```

3.8 Context and Current Node

XSLT distinguishes between the *context node* and the *current node*. The location path `self::node()` selects the context node. The `current()` function returns a node-set that has the current node as its only member. During the execution of a template rule, the current node is the node that "fired" the template rule, i.e., the node for which the template rule was the best match. The current node is almost always the same thing as the context node. However, within square brackets the current node is usually different from the context node. In the following example, the second occurrence of `current()` can be replaced by `self::node()`. However, replacing the first occurrence of `current()` by `self::node()` (or by `self::node()/price/@unit`) changes the meaning of the stylesheet.

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

```
                version="1.0">
   <xsl:template match="/">
      <xsl:apply-templates select="//price/@unit"/>
   </xsl:template>

   <xsl:template match="@unit">
      <BR/>
      <xsl:value-of select="count(/catalog/*[price/@unit=current()])"/>
      vehicle prices are expressed in
      <xsl:value-of select="current()"/>
   </xsl:template>
</xsl:stylesheet>
```

It is an error to use the `current()` function in a pattern.

3.9 Eliminating Duplicates

The XSLT stylesheet of Section 3.8 outputs duplicate rows:

```
<BR />2 vehicle prices are expressed in BEF
<BR />2 vehicle prices are expressed in EUR
<BR />2 vehicle prices are expressed in BEF
<BR />2 vehicle prices are expressed in EUR
```

Can you change the stylesheet so as to avoid duplicate rows?

Hint 4 An easy "trick" is to replace the instruction

```
<xsl:apply-templates select="//price/@unit"/>
```

with the following one:

```
<xsl:apply-templates select="//price/@unit[not(.=preceding::*/@unit)]"/>
```

Can you explain this trick?

3.10 Variables

What is the total value in BEF of all vehicles in the catalog?

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="catalog">
  <xsl:variable name="eurtobef"
                select="round(40.3399*sum(.//price[@unit='EUR']))"/>
  <xsl:variable name="bef"
                select="sum(.//price[@unit='BEF'])"/>
  <xsl:value-of select="$eurtobef+$bef"/>
</xsl:template>
</xsl:stylesheet>
```

Hint 5 The preceding stylesheet can be encoded without variables:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="catalog">
<xsl:value-of select="round(40.3399*sum(.//price[@unit='EUR']))
                      +sum(.//price[@unit='BEF'])"/>
</xsl:template>
</xsl:stylesheet>
```

3.11 Creating Element and Attribute Nodes

Up to now, to create a `mylabel` element node in the output, we simply added the opening tag `<mylabel>` and closing tag `</mylabel>` in the XSLT stylesheet. An "equivalent" way to do this is with `xsl:element`, as follows:

```
<mylabel>                    <xsl:element name="mylabel">
  ...          ⟺              ...
</mylabel>                   </xsl:element>
```

Since the `name` attribute value of `xsl:element` can be set by an expression, this allows for creating element nodes whose names are unknown at the time we write the stylesheet. For example, assume we want to turn text values of car models into element names, as in the following XML document:

```
<LIST-OF-CARS>
  <Renault-CLIO prix="115000"/>
  <Peugeot-Partner prix="12000"/>
</LIST-OF-CARS>
```

Notice that since element names cannot contain spaces, we translated `Renault CLIO` into `Renault-CLIO`. The stylesheet looks as follows:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
   <xsl:element name="LIST-OF-CARS">
      <xsl:apply-templates select="//car"/>
   </xsl:element>
</xsl:template>

<xsl:template match="car">
     <xsl:variable name="var" select="translate(./model,' ','-')"/>
     <xsl:element name="{$var}">
          <xsl:attribute name="prix">
               <xsl:value-of select="price"/>
          </xsl:attribute>
     </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

The text value of each car model is stored in a variable called `var`. The function

```
translate(string1,string2,string3)
```

converts `string1` by replacing the characters in `string2` with the characters in `string3`. The curly braces ({}) in `name="{$var}"` tell the XSLT processor that the value for `name` is the value assigned to the variable `var`, rather than the string `$var`. Notice also how attribute nodes are created by `xsl:attribute`.

Hint 6 I used the variable `$var` in the preceding template rule for readability reasons. Of course, you can eliminate this variable:

```
<xsl:template match="car">
     <xsl:element name="{translate(./model,' ','-')}">
          <xsl:attribute name="prix">
               <xsl:value-of select="price"/>
          </xsl:attribute>
     </xsl:element>
</xsl:template>
```

4 Elaborated Example

Give two tables: the first one containing models and prices of all cars, the second one containing frame heights and prices of all bikes. All prices should be converted into BEF. In the following solution, all HTML markup has been moved to the right to allow focusing on the XSLT stuff.

```
<xsl:stylesheet version="1.0"
                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
               <HTML><BODY> <H1>Cars</H1><TABLE BORDER="3">
               <TR><TH>Car Model</TH><TH>Price</TH></TR>
   <xsl:apply-templates select="//car"/>
               </TABLE><H1>Bikes</H1><TABLE BORDER="3">
               <TR><TH>Frame Height</TH><TH>Price</TH></TR>
   <xsl:apply-templates select="//bike"/>
               </TABLE></BODY></HTML>
</xsl:template>
<xsl:template match="car">
               <TR><TD>
   <xsl:value-of select="model"/>
               </TD><TD>
   <xsl:apply-templates select="price"/>
               </TD></TR>
</xsl:template>
<xsl:template match="bike">
               <TR><TD>
   <xsl:value-of select="height"/>
               </TD><TD>
   <xsl:apply-templates select="price"/>
               </TD></TR>
</xsl:template>
<xsl:template match="price[@unit='BEF']">
   <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="price[@unit='EUR']">
   <xsl:value-of select="round(40.3399*.)"/>
</xsl:template>
</xsl:stylesheet>
```

5 Exercises

5.1 Questions

Fig. 9 shows the tree representation of the following XML document `dic.xml`:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE dict SYSTEM "dic.dtd">
<dict>
  <b>
    <e>
      <word>be</word>
      <a>
        <r>
          <word>bear</word>
        </r>
```

```
      </a>
      <e>
        <word>bee</word>
        <r>
          <word>beer</word>
        </r>
      </e>
    </e>
  </b>
  <e>
    <a>
      <r>
        <word>ear</word>
      </r>
    </a>
    <r>
      <a>
        <word>era</word>
      </a>
    </r>
  </e>
</dict>
```

The tree can be regarded as an ordered prefix tree representation of a dictionary. Each word in the dictionary is constructed by lining up the one-letter tags in a path from the document element node to a `word` element node. The word is repeated in the child text node of the `word` element node. Although this example uses XML in a somehow strange way, it allows many interesting exercises.

Question 1 Write a DTD that captures the intended trees.

Question 2 Describe the result of the transformation of `dic.xml` using the following XSL stylesheet `select.xsl`:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:apply-templates select="//e//word"/>
   </HTML>
</xsl:template>
<xsl:template match="r//word">
   <BR/>
   <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="word"/>
</xsl:stylesheet>
```

Question 3 Same question for:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:apply-templates
       select="//word[following-sibling::*/descendant::word]"/>
   </HTML>
```

```
</xsl:template>
<xsl:template match="word">
   <BR/>
   <xsl:value-of select="."/>
</xsl:template>
</xsl:stylesheet>
```

Question 4 Same question for:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:value-of
       select="//word[count(preceding::word)=count(following::word)]"/>
   </HTML>
</xsl:template>
</xsl:stylesheet>
```

Question 5 Adapt the DTD and the above stylesheets to the tree representation of Fig. 10 where words are "delimited" by attribute nodes.

Question 6 Write two XSL stylesheets that transform either representation into the other. To answer this question, one needs to use the tags `xsl:copy`, `xsl:attribute`, and `xsl:element`.

5.2 Answers

Answer to Question 1.

```
<!ELEMENT dict (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT word (#PCDATA)>
<!ELEMENT a (word?,a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT b (word?,a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT c (word?,a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
...
<!ELEMENT z (word?,a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
```

Answer to Question 5. The XML document `dic2.xml` looks like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE dict SYSTEM "dic2.dtd">
<?xml-stylesheet type="text/xsl" href="select.xsl"?>
<dict>
  <b>
    <e word="be">
      <a>
        <r word="bear"/>
      </a>
      <e word="bee">
        <r word="beer"/>
      </e>
    </e>
  </b>
  <e>
    <a>
      <r word="ear"/>
    </a>
    <r>
      <a word="era"/>
    </r>
  </e>
</dict>
```

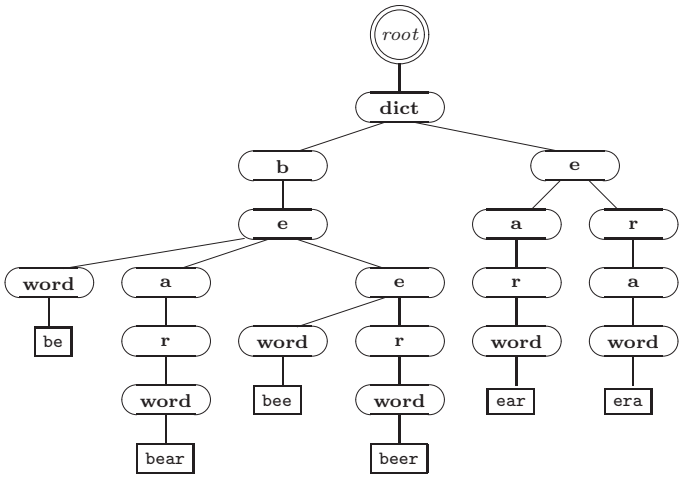


Figure 9: Tree representation of `dic.xml`.

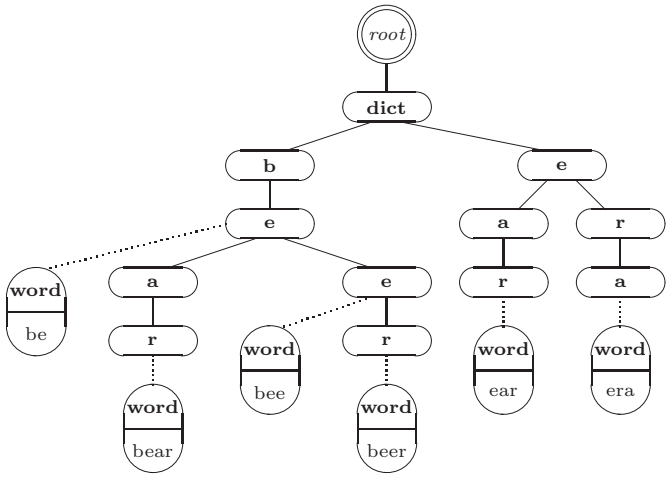


Figure 10: Tree representation of `dic2.xml`.

The DTD now looks as follows:

```
<!ELEMENT dict (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT a (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT b (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ELEMENT c (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
...
<!ELEMENT z (a?,b?,c?,d?,e?,f?,g?,h?,i?,j?,k?,l?,m?,n?,o?,p?,q?,r?,s?,t?,u?,v?,w?,x?,y?,z?)>
<!ATTLIST a word CDATA #IMPLIED>
<!ATTLIST b word CDATA #IMPLIED>
<!ATTLIST c word CDATA #IMPLIED>
...
<!ATTLIST z word CDATA #IMPLIED>
```

The equivalent stylesheet for Question 2 is:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:apply-templates select="//e//@word"/>
   </HTML>
</xsl:template>
<xsl:template match="r//@word">
  <BR/>
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="@word"/>
</xsl:stylesheet>
```

The equivalent stylesheet for Question 3 is:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:apply-templates select="//@word[../descendant::*/@word]"/>
   </HTML>
</xsl:template>
<xsl:template match="@word">
  <BR/>
  <xsl:value-of select="."/>
</xsl:template>
</xsl:stylesheet>
```

The equivalent stylesheet for Question 4 is:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml"/>
<xsl:template match="/">
   <HTML>
      <xsl:value-of select=
      "//*[@word][count(ancestor::*[@word]|preceding::*[@word])=
                  count(descendant::*[@word]|following::*[@word])]/@word"/>
   </HTML>
```

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

Answer to Question 6.

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/|*">
   <xsl:copy>
      <xsl:apply-templates/>
   </xsl:copy>
</xsl:template>
<xsl:template match="word">
   <xsl:attribute name="word">
      <xsl:value-of select="."/>
   </xsl:attribute>
</xsl:template>
</xsl:stylesheet>
```

and

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/|*">
   <xsl:copy>
      <xsl:apply-templates select="*|@*"/>
   </xsl:copy>
</xsl:template>
<xsl:template match="@word">
   <xsl:element name="word">
      <xsl:value-of select="."/>
   </xsl:element>
</xsl:template>
</xsl:stylesheet>
```