universität wien

Data Transformation

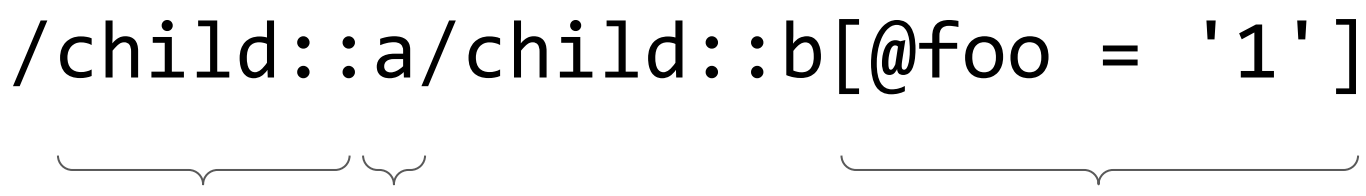
Faculty of Computer Science
Workflow Systems and Technologies

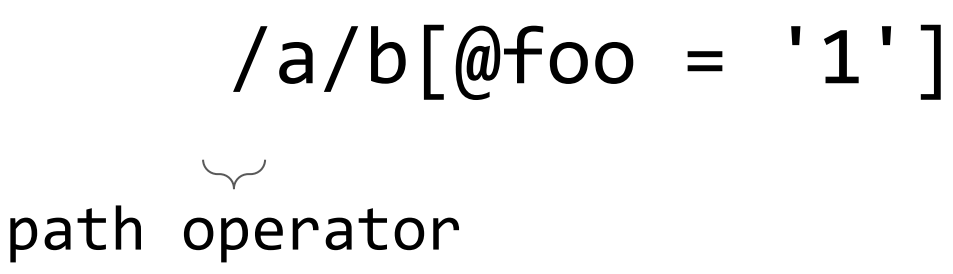
Interoperability

XPath

- Expression language
- Hierarchical addressing of nodes in tree-like structures
- Supports XML and JSON (supports arrays and maps via XDM since version 3.1)
- Data processing and transformation e.g. with XQuery and XSLT
- Support in browsers for DOM access and manipulation in addition to functions such as `document.getElementById()` or `document.querySelector()`
  - *Firefox:* `document.evaluate()`
  - *Chrome:* `$x("")`
- Tool support for XPath features for versions greater 1.0 varies strongly!

Path Expressions





Path Expressions

Path expressions return a set of matching nodes of a tree.

- / connects nodes to form path expressions.
- // expands to match intervening nodes in path expressions.
- * wild card for nodes in path expressions.

Path expressions with '/' or '//' at the start represent absolute path expressions, they select nodes starting from the root of a tree.

A path expression starting with a node is referred to as a relative path expression, it selects nodes relative to the **context node** (which can be any element in the tree).

Path Axes

An axis determines relationships, it locates nodes relative to the context node.

```
ReverseAxis ::= ("parent" "::")
              | ("ancestor" "::")
              | ("preceding-sibling" "::")
              | ("preceding" "::")
              | ("ancestor-or-self" "::")
```

```
ForwardAxis ::= ("child" "::")
              | ("descendant" "::")
              | ("attribute" "::")
              | ("self" "::")
              | ("descendant-or-self" "::")
              | ("following-sibling" "::")
              | ("following" "::")
              | ("namespace" "::")
```

Abbreviated syntax:

- `  -> child::node()`
- `. -> self::node()`
- `.. -> parent::*`
- `// -> /descendant-or-self::node()/`
- `@*, @name -> attribute::*, attribute:name`

Node Tests

Node tests determine which nodes of an axis to select for the step.

Name Test Examples:

- `child::ns:para` - matches nodes named 'para' in namespace 'ns'
- `child::*` - matches any applicable node on the axis wrt. the context node
- `*:para, ns:*` - matching nodes with wildcards using namespaces
- `[name() = 'para']` - matches nodes named 'para' in a predicate
- `attribute::*` - matches all attributes of the context node

Kind Test Examples:

- `node()` - matches all nodes
- `text()` - matches text nodes
- `element()` - matches all element nodes
- `comment()` - matches all comment nodes

Predicates

Predicates are composed of expressions and are applied on the axes and steps, e.g. `child::para[pred1][pred2].` See non-complete list of options:

- Axes and node tests
- Boolean operators (last highest precedence)
  `or, and, {=, !=}, {<=, <, >=, >}`
- Arithmetic operators
  `+ - * div mod`
- Set operator
  `|,union intersect ()`
- Functions
  `number() count() sum()`
  `last() position()`
  `name()`
  `string() substring() starts-with() concat(),||`
  `boolean() true() false() not()`

XPath in Action

```
Ex1  /a/b/c
Ex2  /a/*/c
Ex3  //b/./*
Ex4  //*[@bar]
Ex5  //*[@foo]/*
Ex6  //c/../..//*[@bar]
Ex7  //*[@bar='I' or @bar='II']
Ex8  //*[contains(@bar,'I')]
Ex9  //*[name(.)='c']
Ex10 (//c[1]) | //c[2] | //c
```

```
<a>
  <b foo="1">
    <c bar="I">X</c>
  </b>
  <b foo="2">
    <c bar="II">Z</c>
  </b>
  <b foo="3"/>
</a>
```

Exercises

What are the outcomes of the various axis operators?

- Select the ancestors of the second *b*.
- Select all the siblings after the first *b*.
- Select all the descendants of the first *b*.
- Select *b* with the min value for *foo*. Variation: find the min value only using boolean and comparison operators (i.e. don't use min() or max(), thus only XPath 1.0 features).

Data Transformation

- Transform data from one representation into another representation
  - Between different data formats (e.g. JSON to XML and vice versa)
  - Between different data arrangements of the same data format (e.g. XML to XML, JSON to JSON)
- *XSLT and XQuery* - W3C recommendations
  - Initially introduced for XML, recent versions also support JSON
  - XSLT (Extensible Stylesheet Language Transformation) uses declarative pattern matching.
  - XQuery - a querying language with similarities to SQL that uses expressions to compose queries.

Minimal Examples

XSLT Stylesheet

```
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="3.0">

  <xsl:template match="/">
    <foo/>
  </xsl:template>

</xsl:stylesheet>
```

XQUERY Expression

```
xquery version "3.1";
<foo/>
```

Output

```
<foo/>
```

All XSLT documents must be defined in the root element *stylesheet*. Version declaration signals the targeted feature set. Both examples here produce the same XML output.

XQUERY: Fundamental Building Blocks I

Expression

```
<my-new-xml>
  { for $node in /abc/a
    where $node/@foo = "g"
    return $node                    }
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XQUERY: Fundamental Building Blocks I

Expression

```
<my-new-xml>
  { for $node in /abc/a
    where $node/@foo = "g"
    return $node
}
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

/abc/a, $node/@foo = "g" and $node represent XPath Expressions!
$ is a mandatory prefix to access the value of variables and parameters.
{}marks expressions the XQuery processor should evaluate. (Yes, expressions can be mixed with XML!)

XQUERY: Fundamental Building Blocks I

Expression

```
<my-new-xml>
  { for $node in /abc/a
    where $node/@foo = "g"
    return $node                    }
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`for ... in ...` construct iterates over XML nodes.
`return`s XML after evaluating an expression; use `{}` for expressions if mixed with XML.

XQUERY: Fundamental Building Blocks II

Expression

```
<my-new-xml>
  { for $node in /abc/a
    let $variable := $node
    order by $node/@foo
    return $variable
}
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XQUERY: Fundamental Building Blocks II

Expression

```
<my-new-xml>
  { for $node in /abc/a
    let $variable := $node
    order by $node/@foo
    return $variable                    }
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`let` declares a new variable `$` that can be assigned the result of an expression.
`order by` changes the order in which the nodes returned by `for` are output.

XQUERY: Fundamental Building Blocks III

Expression

```
<my-new-xml>
  { for $parent in /abc/a
    return
      <a>
      { for $child in $parent/*
        return element
                 { $parent/@foo }
                 { attribute
                     {"type"}
                     {$child/name()},
                   $child/text()      } }
      </a>                                }
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a>
    <g type="b">X</g>
    <g type="c">Y</g>
  </a>
  <a>
    <f type="b">M</f>
    <f type="c">N</f>
  </a>
</my-new-xml>
```

XQUERY: Fundamental Building Blocks III

Expression

```
<my-new-xml>
  { for $parent in /abc/a
    return
      <a>
      { for $child in $parent/*
        return element
                 { $parent/@foo }
                 { attribute
                     {"type"}
                     {$child/name()},
                   $child/text()      } }
      </a>                                  }
</my-new-xml>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a>
    <g type="b">X</g>
    <g type="c">Y</g>
  </a>
  <a>
    <f type="b">M</f>
    <f type="c">N</f>
  </a>
</my-new-xml>
```

for expressions can be nested!

XSLT: Fundamental Building Blocks I

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:if test="./@foo = 'g'">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XSLT: Fundamental Building Blocks I

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:if test="./@foo = 'g'">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`/`, `./@foo = 'g'` and `.` represent XPath expressions!
`select` and `match` return a set of nodes.

XSLT: Fundamental Building Blocks I

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:if test="./@foo = 'g'">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`for-each` iterates over the set of nodes obtained from the `select` expression. To access a node in focus use `.`
`if` enables to do conditional processing based on the outcome of the `test` expression.
`copy-of` returns a copy of the node and its children.

XSLT: Fundamental Building Blocks I

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:if test="./@foo = 'g'">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`xsl` refers to the namespace definition which has been excluded here due to the lack of space.
`template` provides access to a subset of the XML document tree. By convention the default template matches the root of the document tree with `template match="/"` and serves as an entry point for processing.

XSLT: Fundamental Building Blocks II

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:sort select="@foo"/>
    <xsl:variable name="var" select="."/>
    <xsl:copy-of select="$var"/>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XSLT: Fundamental Building Blocks II

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="abc/a">
    <xsl:sort select="@foo"/>
    <xsl:variable name="var" select="."/>
    <xsl:copy-of select="$var"/>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`variable` declares a variable using a XPath expression.
`$` followed by the value under `name` as suffix grants access to the node referenced by a `variable`.
`sort` orders nodes by the attribute foo. Note that the order of declarations matter!

XSLT: Fundamental Building Blocks III

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="./abc/a">
   <xsl:variable name="parent" select="."/>
   <a>
    <xsl:for-each select="$parent/*">
     <xsl:variable name="child" select="."/>
     <xsl:element name="{$parent/@foo}">
      <xsl:attribute name="type">
       <xsl:value-of select="$child/name()"/>
      </xsl:attribute>
      <xsl:value-of select="$child"/>
     </xsl:element>
    </xsl:for-each>
   </a>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a>
    <g type="b">X</g>
    <g type="c">Y</g>
  </a>
  <a>
    <f type="b">M</f>
    <f type="c">N</f>
  </a>
</my-new-xml>
```

XSLT: Fundamental Building Blocks III

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="./abc/a">
   <xsl:variable name="parent" select="."/>
   <a>
    <xsl:for-each select="$parent/*">
     <xsl:variable name="child" select="."/>
     <xsl:element name="{$parent/@foo}">
      <xsl:attribute name="type">
       <xsl:value-of select="$child/name()"/>
      </xsl:attribute>
      <xsl:value-of select="$child"/>
     </xsl:element>
    </xsl:for-each>
   </a>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a>
    <g type="b">X</g>
    <g type="c">Y</g>
  </a>
  <a>
    <f type="b">M</f>
    <f type="c">N</f>
  </a>
</my-new-xml>
```

`value-of` extracts the content of a node.
`{}` evaluates an expression.
`element` and `attribute` allow dynamic node creation.

XSLT: Fundamental Building Blocks VII

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="//a">
   <xsl:call-template name="a"/>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>

<xsl:template name="a">
 <xsl:if test="./@foo = 'g'">
  <xsl:copy-of select="."/>
 </xsl:if>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XSLT: Fundamental Building Blocks VII

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
  <xsl:for-each select="//a">
   <xsl:call-template name="a"/>
  </xsl:for-each>
 </my-new-xml>
</xsl:template>

<xsl:template name="a">
 <xsl:if test="./@foo = 'g'">
  <xsl:copy-of select="."/>
 </xsl:if>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

Named `templates` promote modularity. `call-template` enables to explicitly apply a named `template` to a context node. The called template could again call a template on a set of selected nodes creating an explicit nested chain of transformations on sub-trees.

Declarative and Recursive Template Matching

XSLT: Fundamental Building Blocks IV

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
   <xsl:apply-templates/>
 </my-new-xml>
</xsl:template>

<xsl:template match="a">
 <xsl:if test="./@foo = 'g'">
  <xsl:copy-of select="."/>
 </xsl:if>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

XSLT: Fundamental Building Blocks IV

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
   <xsl:apply-templates/>
 </my-new-xml>
</xsl:template>

<xsl:template match="a">
 <xsl:if test="./@foo = 'g'">
  <xsl:copy-of select="."/>
 </xsl:if>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

`apply-templates` attempts to apply matching `template` to all children of the context node. *Q: What happens if a matching node also has descendants that would match a template's pattern? Also, what happens if no matching template is found?*

XSLT: Fundamental Building Blocks VI

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
   <xsl:apply-templates/>
 </my-new-xml>
</xsl:template>

<!-- RULE_TEXT
<xsl:template match="text()">
  <xsl:value-of select="."/>
</xsl:template> -->
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>

    X
    Y


    M
    N

</my-new-xml>
```

A template determines the processing steps for a node it matches. If no template matches, *built-in template rules* are applied that keep the processing of the tree going. The default mode for *on-no-match* processing is *text-only-copy* which recursively matches and returns text nodes found in the sub-tree of a context node. A part of the implementation of this 'back-up' rule can be expressed as the fragment marked as `RULE_TEXT`. *Q: What is the reason for the white space in the output?*

XSLT: Fundamental Building Blocks VI

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
   <xsl:apply-templates/>
 </my-new-xml>
</xsl:template>

<!-- Overriding RULE_TEXT -->
<xsl:template match="text()">
  <o><xsl:value-of select="."/></o>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml><o>
  </o><o>
    </o><o>X</o><o>
    </o><o>Y</o><o>
  </o><o>
  </o><o>
    </o><o>M</o><o>
    </o><o>N</o><o>
  </o><o>
</o></my-new-xml>
```

XSLT: Identity Transformation

Stylesheet

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

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

XSLT: Fundamental Building Blocks V

Stylesheet

```
<xsl:template match="/">
 <my-new-xml>
   <xsl:apply-templates select="//a"/>
 </my-new-xml>
</xsl:template>

<xsl:template match="a">
 <xsl:if test="./@foo = 'g'">
  <xsl:copy-of select="."/>
 </xsl:if>
</xsl:template>
```

Input

```
<abc>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
  <a foo="f">
    <b>M</b>
    <c>N</c>
  </a>
</abc>
```

Output

```
<my-new-xml>
  <a foo="g">
    <b>X</b>
    <c>Y</c>
  </a>
</my-new-xml>
```

The application of `apply-templates` can also be limited to a specific set of nodes using `select`.

JSON Transformation with XSLT and XQuery

- Parse JSON into the XDM representation and use maps and arrays to access and extract parts. However, more complex for transformation tasks (see paper for examples).
- Alternative: conversion of JSON into a lossless representation in XML; enables to apply existing transformation strategies and convert final result back to JSON (see next slides).
- **Exercise:** Create a generic language in XML that represents all required constructs of JSON. The XML representation must allow for conversions between XML and JSON without loss of information. Following relationship must hold true: `xml2json(json2xml(json_payload)) == json_payload.`

XSLT: JSON to XML Conversion From a Variable

Stylesheet

```
<xsl:variable name="json">
 <!--Replace this comment with Input on the rhs.-->
</xsl:variable>

<xsl:output method="xml" indent="yes"/>

<xsl:template name="xsl:initial-template">
  <xsl:copy-of select="json-to-xml($json)"/>
</xsl:template>
```

Input

```
{
  "abc": [
    { "foo" : "g",
      "a": {
        "b": "X",
        "c": "Y" }},
    { "foo" : "f",
      "a": {
        "b": "M",
        "c": "N" }}
  ]
}
```

Output

```
<map
xmlns="http://www.w3.org/2005/xpath
-functions">
 <array key="abc">
  <map>
   <string key="foo">g</string>
   <map key="a">
    <string key="b">X</string>
    <string key="c">Y</string>
   </map>
  </map>
  <map>
   <string key="foo">f</string>
   <map key="a">
    <string key="b">M</string>
    <string key="c">N</string>
   </map>
  </map>
 </array>
</map>
```

XSLT: JSON to XML Conversion From a File

Stylesheet

```
<xsl:param name="json-file"/>

<xsl:output method="xml" indent="yes"/>

<xsl:template name="xsl:initial-template">
  <xsl:copy-of
select="json-to-xml(unparsed-text($json-file))"/>
</xsl:template>
```

unparsed-text reads the file passed to the processor at runtime via the parameter 'json-file' and json-to-xml parses JSON into an intermediate lossless XML representation (note the associated namespace in the output).

Input

```
{
  "abc": [
    { "foo" : "g",
      "a": {
        "b": "X",
        "c": "Y" }},
    { "foo" : "f",
      "a": {
        "b": "M",
        "c": "N" }}
  ]
}
```

Output

```
<map
xmlns="http://www.w3.org/2005/xpath
-functions">
 <array key="abc">
  <map>
   <string key="foo">g</string>
   <map key="a">
    <string key="b">X</string>
    <string key="c">Y</string>
   </map>
  </map>
  <map>
   <string key="foo">f</string>
   <map key="a">
    <string key="b">M</string>
    <string key="c">N</string>
   </map>
  </map>
 </array>
</map>
```

XSLT: JSON to JSON - Complete Example

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xpath-default-namespace="http://www.w3.org/2005/xpath-functions"
  xmlns:fn="http://www.w3.org/2005/xpath-functions"
  version="3.0">

  <xsl:param name="json-file"/>
  <xsl:output method="text" indent="yes"/>

  <xsl:template name="xsl:initial-template">
    <xsl:variable name="xml" select="json-to-xml(unparsed-text($json-file))"/>
    <xsl:variable name="transformed-xml">
      <fn:array>
        <xsl:for-each select="$xml//array[@key='abc']/map">
          <xsl:call-template name="foo-map"/>
        </xsl:for-each>
      </fn:array>
    </xsl:variable>

    <!-- Comment out line below when output method is set to xml (debugging)-->
    <xsl:value-of select="xml-to-json($transformed-xml)"/>

    <!-- Comment out line below when output method is set to text -->
    <!-- <xsl:copy-of select="$transformed-xml"/> -->
  </xsl:template>

  <xsl:template name="foo-map">
    <fn:map>
      <fn:array key="{./string/text()}">
        <fn:map>
          <xsl:copy-of select="./map/string"/>
        </fn:map>
      </fn:array>
    </fn:map>
  </xsl:template>
</xsl:stylesheet>
```

Input

```
{
  "abc": [
    { "foo" : "g",
      "a": {
        "b": "X",
        "c": "Y" }},
    { "foo" : "f",
      "a": {
        "b": "M",
        "c": "N" }}
  ]
}
```

Output

```
[
  {
    "g": [
      {
        "b": "X",
        "c": "Y"
      }
    ]
  },
  {
    "f": [
      {
        "b": "M",
        "c": "N"
      }
    ]
  }
]
```

XQuery: JSON to JSON - Complete Example

```
xquery version "3.1";
declare namespace output =
"http://www.w3.org/2010/xslt-xquery-serialization";
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare namespace fn = "http://www.w3.org/2005/xpath-functions";
declare option output:method "text";
declare option output:indent "yes";
declare option output:omit-xml-declaration "yes";
declare variable $json-file as xs:string external;

xml-to-json(
<fn:array>
{for $map in
json-to-xml(unparsed-text($json-file))//fn:array[@key='abc']/fn:map
  return
    <fn:map>
      <fn:array key="{$map/fn:string/text()}">
        <fn:map>
          {$map/fn:map/fn:string}
        </fn:map>
      </fn:array>
    </fn:map>}
</fn:array>
)
```

Input

```
{
  "abc": [
    { "foo" : "g",
      "a": {
        "b": "X",
        "c": "Y" }},
    { "foo" : "f",
      "a": {
        "b": "M",
        "c": "N" }}
  ]
}
```

Output

```
[
  {
    "g": [
      {
        "b": "X",
        "c": "Y"
      }
    ]
  },
  {
    "f": [
      {
        "b": "M",
        "c": "N"
      }
    ]
  }
]
```

Exercises

- Try to run the provided examples. Download the provided script and run:

```
./iopdst.sh create
./iopdst.sh shell

# in the shell
./iopdst.sh xslt30-xml data.xml xml.xslt out.xml
./iopdst.sh xslt30-json data.json json.xslt out.json
./iopdst.sh xquery31-xml data.xml xml.xq out.xml
./iopdst.sh xquery31-json data.json json.xq out.json
```

- Convert XML into JSON and vice versa.
- For additional commands such as for conditional processing, see the reference for XSLT here and XQuery here.

Additional Topics

- Sometimes data doesn't fit in memory or data is received as a stream; XSLT provides capabilities for processing streams (see documentation).
- JSONiq - an alternative querying language for JSON inspired by XQuery (see the specification and paper).
- JSONPath - an attempt to define a similar language to XPath for JSON. See the draft.
- Support for JSON trees with XPath 3.1 (see specification).