

# Compilers and Language Processing Tools

Summer Term 2011

Prof. Dr. Arnd Poetzsch-Heffter

Software Technology Group  
TU Kaiserslautern



## Content of Lecture

1. Introduction
2. Syntax and Type Analysis
  - 2.1 Lexical Analysis
  - 2.2 Context-Free Syntax Analysis
  - 2.3 Context-Dependent Analysis
3. Translation to Target Language
  - 3.1 Translation of Imperative Language Constructs
  - 3.2 Translation of Object-Oriented Language Constructs
4. Selected Topics in Compiler Construction
  - 4.1 Intermediate Languages
  - 4.2 Optimization
  - 4.3 Register Allocation
  - 4.4 Just-in-time Compilation
  - 4.5 Further Aspects of Compilation
5. Garbage Collection
6. XML Processing (DOM, SAX, XSLT)

# 6. XML Processing

## Chapter Outline

- 6.1 Introduction
- 6.2 Specification of XML Documents
  - 6.2.1 DTD
  - 6.2.2 XSD
- 6.3 Processing XML Documents
  - 6.3.1 SAX
  - 6.3.2 DOM
- 6.4 Transforming XML Documents
  - 6.4.1 XSLT
  - 6.4.2 XPath
  - 6.4.3 More Complex XSLT
- 6.5 Concluding Remarks

## 6.1 Introduction

## XML- eXtensibleMarkup Language

XML is a standard for structuring and storing and information.  
(XML is **not** a programming language).

### Characteristics of XML:

- XML documents are text files with Unicode character encoding.
- XML separates data from its representation.
- XML structures data in a tree-like way by markup.
- XML does not define a semantics of the data itself.
- XML is extensible in order to create new languages.

## Example XML Document

```
<?xml version="1.0"?>
<!-- file people.xml -->
<people>
  <person>
    <name>Alan Turing</name>
    <profession>computer scientist</profession>
  </person>
  <person>
    <name>Richard M. Feynman</name>
    <profession>physicist</profession>
  </person>
</people>
```

## Why XML ...?

- Widely used and universally available
- Simplifies exchanging information between incompatible systems and via the internet
- Standardized format
- Platform-independent
- Many tools are available for XML processing.

## Examples of XML-based Languages

- MathML (Mathematical Markup Language)
- WAP (Wireless Application Protocol)
- RDF (Resource Description Framework)
- SVG (Scaleable Vector Graphics)
- HTML (Hypertext Markup Language)
- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Description Language)
- ODF (Open Document Format)
- RSS (RDF Site Summary)

## 6.2 Specification of XML Documents

## XML Documents

- XML documents may have an XML declaration (optional)  
`<?xml version="1.0" encoding="UTF-8"?>`
- There is exactly one root element.
- Each element has a matching start and end tag.
- Elements can be nested.
- Empty elements use a special notation.
- Elements can be repeated (distinguished by position).
- Elements can have attributes (zero to any number).
- Attributes can only occur once in elements.
- Comments cannot be placed inside tags.

## XML Elements

- Element names consist of letters, numbers, ".", "\_", "-", ":" and start with letter or "\_".
- Element names are case-sensitive:  
`<firstname>` and `<FirstName>` are different
- Start tag encloses element name in `< >`, e.g., `<firstname>`
- Matching end tag: `</firstname>`
- Empty element: `<firstname/>`

## Attributes

- Attributes carry additional information of elements that is not considered content (e.g., metadata).
- Attributes are specified in tags:  
`<name language="DE"> Chris </name>`
- Attribute values are enclosed in quotes and may not contain "<"

## Grammar for XML documents (Part)

According to <http://www.w3.org/TR/xml11/>

```

document ::= ( prolog element Misc* )
prolog ::= XMLDecl Misc* (doctypeddecl Misc*)?
XMLDecl ::= '<?xml' VersionInfo EncodingDecl? '?>'
Misc ::= Comment | PI | S

element ::= EmptyElemTag
          | STag content ETag

EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'
STag ::= '<' Name (S Attribute)* S? '>'
Attribute ::= Name Eq AttValue
ETag ::= '</' Name S? '>'

```

PI processing instructions; S whitespace

## XML Document Example

```
<?xml version="1.0" encoding="UTF-8"?>
<section id="xml" author="bob">
  <title> Extensible Markup Language (XML)</title>
  <p> XML is based on SGML (Section <ref name="sgml"/>) </p>
  <p type="example">XML can be used ... </p>
  <section id="xml-syntax">
    <title> XML Syntax</title>
    <p>Section <ref name="sgml-syntax"/> ...</p>
  </section>
</section>
```

## Mixed Content

**Mixed content** refers to elements which have text content mixed with elements.

### Example:

```
<p>The term <em>Mixed content</em> in XML refers to elements
<a href="http://www.w3.org/TR/xml/#sec-mixed-content">
which have text content mixed with elements</a>. The elements
are on the same level as the text nodes of the
mixed content.</p>
```



## Well-formed and Valid XML Documents

- **Well-formed** documents satisfy all basic constraints of the XML specification:
  - ▶ they can be parsed according to the XML grammar
  - ▶ they satisfy the additional constraints (e.g., matching start and end tags)
  - ▶ this means they can be translated into a tree
- **Valid** documents must satisfy additional constraints (e.g., of an DTD, XSD, ...)
  - ▶ a document must be well-formed before it can be validated
  - ▶ all used elements and attributes in the document have been defined and are used according to their definition.

## Specifying Valid XML Documents

- Document Type Definitions (DTD)
  - ▶ Part of SGML, developed before XML
  - ▶ Not itself in XML syntax, but included in XML definition
  - ▶ No type system
- XML Schemas (XSD)
  - ▶ Developed after XML
  - ▶ More complicated than DTD
  - ▶ In XML syntax
  - ▶ Support namespaces to resolve conflicts between element and attribute names
  - ▶ Elaborated and extensible type system

## 6.2.1 DTD

## DTD by Example

### Example DTD

```
<!ELEMENT address (name, phone*)>
<!ELEMENT name (#PCDATA)>

<!ELEMENT phone (#PCDATA)>
<!ATTLIST phone type ( voice | fax ) #REQUIRED>
```

### Example Document

```
<address>
  <name> MaxMuster </name>
  <phone type = "voice"> 4711 </phone>
  <phone type = "fax"> 4712 </phone>
</address>
```

## 6.2.2 XSD

## XML Namespaces

- Namespaces are used to resolve conflicts between element and attribute names if two XML formats are merged.
- A namespace is defined by a **Uniform Resource Identifier (URI)**:
  - ▶ Default Namespace:  
`html xmlns="http://www.w3.org/1999/xhtml"`
  - ▶ Prefix Namespace:  
`xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml"`
- If an element name has no prefix, it is in the default namespace.
- If a name has a prefix, the prefix must be bound to a namespace (qualified names).
- A prefix is only valid inside of the element defining the prefix.

## Uniform Resource Identifier

URIs identify resources and are typically hierarchically structured.

### Commonly used URIs:

- Uniform Resource Locators (URLs): define resource location and access protocol, e.g., `http://www.w3.org/1999/xhtml`
- Uniform Resource Names (URNs): an identifying (existing) name, e.g., `urn:ietf:rfc:2648`

List of possible URI schemes:

`http://www.iana.org/assignments/uri-schemes.html`

## Namespaces by Example

```
<catalog xmlns:dc="http://purl.org/dc"
          xmlns:pt="http://vangoghgallery.com/pt" >
  <dc:description>
    <dc:title>Impressionist Paintings</dc:title>
    <dc:creator>Elliotte Rusty Harold</dc:creator>
  </dc:description>
  <pt:painting>
    <pt:title>Memory of the Garden at Etten</pt:title>
    <pt:artist>Vincent van Gogh</pt:artist>
  </pt:painting>
</catalog>
```

## XML Schema - XSD

- is an XML dialect itself
- supports namespaces
- sophisticated extensible type system
- allows rigorous automated checking of document content
- enables automated conversion of XML documents, e.g., to Java Objects and vice versa

## Structure of XSD

A schema defines

- which elements a document may contain
- which attributes these elements have
- the relationship between parent and child elements
- the ordering of elements
- the number of child elements
- if elements are empty or may contain text
- data types for elements and attributes
- default values and constants for elements and attributes

## Types in XSD

- Atomic types: integer, boolean, string, ... (more built-in types)
- Simple types : derived from atomic types by list and union operators
- Complex types
  - ▶ can contain elements and attributes themselves
  - ▶ ordered and unordered sequences of elements
  - ▶ selection of elements
  - ▶ constraints on values of elements and attributes
  - ▶ constraints on occurrences of elements and attributes

## Selected XSD Language Elements

- Simple elements:  
`<xs:element name="lastName" type="xs:string"/>`  
`<xs:element name="age" type="xs:integer"/>`

Instance:

```
<lastName> Schmid </lastName>
```

```
<age> 34 </age>
```

## Selected XSD Language Elements (2)

- Elements with attributes:

```
<xs:element name="shoeSize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:positiveInteger">
        <xs:attribute name="country" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Instance:

```
<shoeSize country="france">35</shoeSize>
```

## Selected XSD Language Elements (3)

- Empty elements:

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="prodId" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
```

Instance:

```
<product prodId="1345"/>
```

## Selected XSD Language Elements (4)

- Elements with a sequence of child elements:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstName" type="xs:string"
        minOccurs="0" maxOccurs="2" />
      <xs:element name="lastName" type="xs:string"
        minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Instance:

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
</person>
```

## Selected XSD Language Elements (5)

- Elements with an unordered sequence of child elements:  
(maxOccurs = 1 for each child element)

```
<xs:element name="person">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```



## Selected XSD Language Elements (6)

- Elements with a selection of child elements:

```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employeeType"/>
      <xs:element name="member" type="memberType"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Each person is either an employee or a member, but not both.

## XSD Example

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.note.org">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:id"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

## XSD Example (2)

Instance:

```
<?xml version="1.0"?>
<note xmlns="http://www.note.org" id = "4711" >
  <to> Bill </to>
  <from> Jani </from>
  <heading> Reminder </heading>
  <body>
    Don't forget the book!
  </body>
</note>
```

## Validation of XML Documents

- Lexical and context-free analysis
- Well-formedness checks (wrt. XML grammar)  
(matching tags, unique attribute names, ...)
- Validation checks for structural integrity  
(is the document schema-valid?)
  - ▶ checking elements and attributes for correct application
  - ▶ checking element contents and attribute values for compliance to constraints

## XSD: Unordered List with Multiple Occurrences

```
<xs:element name="root">
  <xs:complexType>
    <xs:sequence>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="A"/>
        <xs:element name="B"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## XSD: Unordered List with Multiple Occurrences (2)

### Instances:

```
<root>
  <A/>
  <B/>
  <B/>
</root>
```

or

```
<root>
  <B/>
  <A/>
  <A/>
  <B/>
</root>
```

## 6.3 Processing of XML Documents

## Processing XML Documents

1. Map XML documents to programming language data types (XML Data Binding, e.g., JAXB)
2. Function libraries providing XML processing facilities
  - ▶ SAX (Simple API for XML) as an event-based API
  - ▶ DOM (Document Object Model) as a tree-based API
3. XML processing languages (XSLT, XPATH, XQuery, ... )

## 6.3.1 SAX

# SAX

- an event-based API for accessing XML documents
- use event handlers for parsing-related events
- the parser reads a document and raises events when markup is recognized
- for each recognized structure, an event triggers a user-supplied function

## SAX (2)

- Pros:
  - ▶ requires little memory and can handle very large documents
- Cons:
  - ▶ does not allow random access or backward movement
  - ▶ context and history has to be managed by the application
  - ▶ at a certain complexity, SAX parsing requires a lot of additional code

## SAX Parsing

Observers of a SAX Parser must implement the `org.xml.sax.ContentHandler` interface.

The class `org.xml.sax.helpers.DefaultHandler` contains a default implementation for the methods of `ContentHandler` such that observers only have to implement the used methods.

## SAX Parsing (2)

Example for Observer:

```
class mySaxApp extends DefaultHandler{

public void startDocument ()
    { System.out.println("Start document"); }

public void endDocument ()
    { System.out.println("End document"); }

public void startElement (String uri, String name,
                          String qName, Attributes atts) { ... }

public void endElement (String uri, String name, String qName){...}

public void characters (char ch[], int start, int length){...}
}
```

## SAX Parsing (3)

Example application of SAX parser:

```
public static void main (String args[])
    { // create XML Reader
      XMLReader xr = XMLReaderFactory.createXMLReader();

      // create observer
      MySAXApp handler = new MySAXApp();

      // register observer
      xr.setContentHandler(handler);

      // parse document
      InputSource r = new InputSource(FileReader(args[0]));
      xr.parse(r);
    }
```

## 6.3.2 DOM

## DOM

- a tree-based API for accessing XML documents
- based on an in-memory representation of an XML document
- access to tree structure by special methods
- Pros:
  - ▶ random document access
  - ▶ more specific tasks possible, e.g., getting an element's attribute by name
- Cons:
  - ▶ very large documents may not fit into memory
  - ▶ for isolated tasks, the parsing overhead is prohibitive



## DOM Parsing

For representing XML documents as trees, a general data structure is used. Central data type is Node:

```
interface Node {
Node appendChild(Node newChild) ;
Node cloneNode(boolean deep);
NamedNodeMap getAttributes();
NodeList getChildNodes();
Node getFirstChild();
Node getLastChild();
String getLocalName();
String getNamespaceURI();
Node getNextSibling();
String getNodeName();
short getNodeType();
String getNodeValue();
... // und viele Methoden mehr
}
```

## DOM Parsing (2)

The type Document represents complete XML documents:

```
interface Document extends Node { ... }
```

## DOM Parsing (3)

Example application of DOM Parser:

```
public static void main(String[] args) {
    InputSource src = new InputSource(new FileInputStream(args[0]));

    // create parser
    DOMParser prsr = new DOMParser();

    // start parsing
    prsr.parse( src );

    // access to DOM tree
    Document doc = prsr.getDocument();

    // further processing, e.g.
    System.out.print(doc.getFirstChild.getNextSibling.getNodeName());
}
```

## 6.4 Transforming XML Documents

# XSL - eXtensible Stylesheet Language

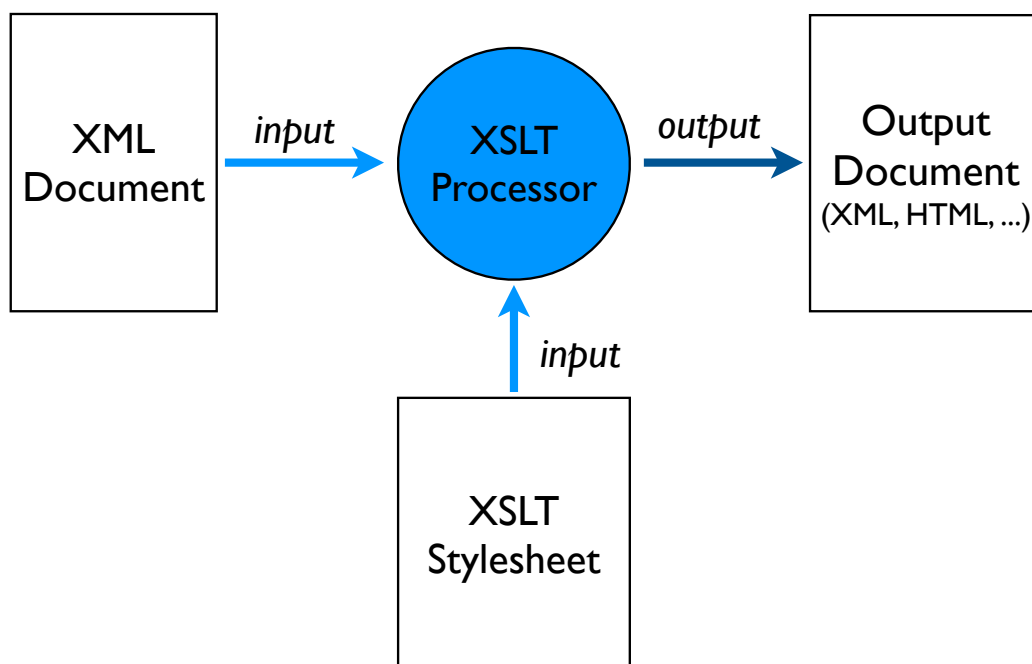
- XSLT: Language for transforming XML
- XPath: Language for selecting XML nodes in an XML document
- XSL-FO (Formatting Objects): Language for describing formatted output

## 6.4.1 XSLT

# XSLT

- XML-based processing language
- not designed for large programming tasks
- standard language for XML-to-XML/text transformations
- based on XPath (navigation and computation)
- needs XSLT processor,  
e.g., `xsltproc -o outputfile xslt_script input.xml`

# XSLT processing



## Examples of XSLT Applications

- Transforming XML to (X)HTML for web servers
- Add content to XML documents, e.g., table of contents
- Transform one XML format to another, e.g., DocBook to ODF/ODT

## Main concepts of XSLT

- XSLT is a functional processing/programming language
- XSLT has built-in behavior for tree traversal
- XSLT defines a set of templates
- A template consists of:
  1. a pattern which selects a set of nodes
  2. a set of actions to perform on these nodes
  3. further processing instructions

## Structure of XSLT stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/> // Alternatives are "xml" or "text"

  <xsl:template match = [Set of Nodes] >
    [Actions]
    // recursively apply matching templates of children
    <xsl:apply-templates/>
  </xsl:template>

</xsl:stylesheet>
```

## Execution of XSLT Stylesheet

Execution starts with root node of input document.

For each node:

1. Find matching templates
2. If matching template exists:
  - ▶ execute actions of template and/or
  - ▶ execute `<xsl:apply-templates/>` recursively
3. If no matching template exists, apply default rules, if applicable

## XSLT default rules

- Root and Element Nodes: recurse on children

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

- Text Node and Attributes: copy text content to output tree

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

- Comments and Processing Instructions are removed from output.

## XSLT default rules (2)

Stop default behavior by introducing matching rule  
overriding the default behavior, e.g.,

```
<xsl:template match="text()" />
```

or by qualifying nodes to which templates are applied by

```
<xsl:apply-templates select= [Set of Nodes] />
```

## XSLT by Example

### Transform XML to HTML

#### Input document:

```
<?xml version="1.0"?>
  <people>
    <person>
      Alan Turing
    </person>
    <person>
      Kurt Goedel
    </person>
    <person>
      John von Neumann
    </person>
  </people>
```

## XSLT by Example (2)

#### XSLT Stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="people">
  <html> <head> People List </head> <body>
    <ul> <xsl:apply-templates/> </ul>
  </body> </html>
</xsl:template>

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



## XSLT by Example (3)

### Output document:

```
<html>
<head> People List </head>
<body><ul>
  <li>
    Alan Turing
  </li>
  <li>
    Kurt Goedel
  </li>
  <li>
    John von Neumann
  </li>
</ul></body>
</html>
```

## 6.4.2 XPath

# XPath

- Language for selecting nodes in an XML document
- Syntax is similar to file system addressing.
- XPath expressions define patterns matching a set of document nodes.
- Matching depends on current evaluation context wrt current node.

# XPath Examples

- `name`: all `<name>` children of current node
- `name/firstName`: all `<firstName>` children of all `<name>` children of current node
- `/`: the root node of the document
- `..`: the current node
- `..`: the parent of the current node
- `//name`: all `<name>` elements in the document
- `..//name`: all `<name>` successors (direct or indirect) of the current node

## More XPath Expressions

- An XPath is absolut, if it starts with "/",  
e.g., /person
- A path can contain a number of steps separated by "/",  
e.g., people/person/born
- Attributes are marked by "@",  
e.g., born/@addressRef
- XPaths can be combined by "|" denoting alternatives,  
e.g., name/firstName| profession | hobby

## Wildcards

XPaths can contain wildcards:

- \* - selects all elements
- @\* - selects all attributes
- node() - selects all nodes
- text() - selects text nodes

Examples:

- // \* - all elements in the document
- / \* - all children of root
- str / \* / title - all <title> grand-children of str elements
- // \* / @ id - all attributes with name id

## XPath Predicates

Predicates can be used to define additional constraints on the values of attributes and elements.

Examples:

- `person[name/firstName!='Alan' ]` - all person children where the first name in the name element is not 'Alan'
- `died[ @addressRef ]` - all <died> children with addressRef attribute
- `born[ @addressRef = 'gbc' ]` - all <born> children with addressRef attribute with value 'gbc'
- `person [born/@date > 1900 ]` - all persons born later then 1900

## XPath Axes

Instead of the abbreviated notation, axes denoting the access direction can be denoted explicitly.

Some examples:

- `self::node()` - the current node itself
- `child::name` - all <name> children
- `parent::node()` - the parent node of the current node
- `descendant::name` - all descendants of <name> node
- `ancestor-or-self::name` - all precessors of <name> node including the <name> node itself
- `following::name` - all nodes following a <name> node in the document text
- `preceding-sibling::name` - the sibling before a <name> node

## XPath functions

XPath has a large set of built-in functions (even more in XPath 2.0) that can be used in XPath predicates and in XSLT scripts for computing values from document nodes.

Some examples:

- Arithmetic, relational and logical operators, e.g., +, -, =, !=, &lt;, and, ...
- String operations, e.g., concat, substring, normalize-space, ...
- `count(//person)` - counts number of person nodes in the document
- `sum(//number)` - computes the sum of the values in `<number>` nodes
- `person[1]` - the first person element
- `person[last()]` - the last person element

### 6.4.3 More Complex XSLT

## Selected XSLT language constructs

- `<xsl:apply-templates ... >` applies templates in style sheet to further nodes.

The next nodes to be processed can be explicitly selected by XPath expression:

```
<xsl:apply-templates select=[XPath expression]/>
```

`<xsl:apply-templates/>` applies templates to all children of current node.

## Selected XSLT language constructs (2)

- `<xsl:for-each ... >` processes template for each node with matching expression:

```
<xsl:template match="people">
  <xsl:for-each select=[XPath expression]>
    [Action]
  </xsl:for-each>
</xsl:template>
```

## Selected XSLT language constructs (3)

- `<xsl:value-of ... ">` selects (and processes) particular nodes in an XML document:

```
<xsl:value-of select="lastName"/>
```

```
<xsl:value-of select="born/@date"/>  
// outputs data attribute
```

```
<xsl:value-of select="sum(//number)"/>  
// sum of all number values
```

```
<xsl:value-of select="count(//person)"/>  
// number of person nodes in document
```

## Selected XSLT language constructs (4)

- `<xsl:call-template ... >` calls a template by its name:

```
<xsl:template match="people" >  
  <xsl:call-template name="templ1"/>  
</xsl:template>
```

```
<xsl:template name=" templ1">  
  <xsl:apply-templates select="person"/>  
</xsl:template>
```

## Selected XSLT language constructs (5)

- Templates with modes allow context-dependent execution of templates:

```
<xsl:template match="people">
  <h4>Table of Contents</h4>
  <xsl:apply-templates select="person" mode="toc"/>
  <xsl:apply-templates select="person"/>
</xsl:template>
```

```
<xsl:template match="person" mode="toc">
  <xsl:value-of select="name/lastName"/>
</xsl:template>
```

```
<xsl:template match="person">
  <xsl:apply-templates select="name"/>
</xsl:template>
```

## Conditional Processing

- `<xsl:if test = [condition]>` allows processing nodes differently:

```
<xsl:template match="person">
  <xsl:if test="born/@date > 1900
    and name/firstName != 'Alan'">
    <xsl:apply-templates select="name"/>
  </xsl:if>
</xsl:template>
```



## Conditional Processing (2)

- `<xsl:choose>` can be used as an if-then-else or as a switch construct:

```
<xsl:choose>
  <xsl:when test="...">
    [Actions if condition is true]
  </xsl:when>
  <xsl:when test="...">
    [Actions if condition is true]
  </xsl:when>
  <xsl:otherwise>
    [Default actions]
  </xsl:otherwise>
</xsl:choose>
```

## Conditional Processing (3)

- `<xsl:sort ...>` allows sorting of nodes during transformation inside `<xsl:apply-templates ...>` or `<xsl:for-each ...>` elements:

```
<xsl:apply-templates select="person">
  <xsl:sort select="name/lastName" order="ascending"/>
</xsl:apply-templates>
```

## Variables and Parameters

In XSLT, also variables can be defined by

```
<xsl:variable name="variablename">
  [Definition of value_of_variable] // can be complex content
</xsl:variable>
```

Value of variable can be read by \$variablename in

```
<xsl:value-of select="$variablename"/>.
```

The value cannot be changed after definition of a variable.

## Variables and Parameters (2)

Example: Computation of a weighed sum

```
<xsl:variable name="tmp">
  <tmpPrice>
    <xsl:for-each select="book">
      <item>
        <xsl:value-of select="price * quantity"/>
      </item>
    </xsl:for-each>
  </tmpPrice>
</xsl:variable>

<xsl:value-of select="sum($tmp/tmpPrice/item)"/>
```

## Variables and Parameters (3)

Declaration of template with parameter:

```
<xsl:template name="myTemplate">
  <xsl:param name="p1"/> // formal parameter declaration
  [Actions using $p1]
</xsl:template>
```

Calling a template with a parameter:

```
<xsl:template match="person">
  <xsl:call-template name="myTemplate">
    <xsl:with-param name="p1"> // current parameter
      <xsl:value-of select="someValue"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

## 6.5 Concluding Remarks

## What is not covered

- XQuery: Querying XML documents
- XSL-FO: Formatting Objects
- Other Schema Languages: RELAX NG, Schematron
- XML Data Binding: JAXB, Castor

## Literature

- Recommendations of the W3C <http://www.w3c.org>
- Many other XML-related resources are available on the web.
- There are many XML books available.  
Make sure you get the newest edition!