Practical Transformation Using XSLT and XPath (XSL Transformations and the XML Path Language)

Crane Softwrights Ltd.
http://www.CraneSoftwrights.com

+//ISBN 978-1-894049::CSL::Courses::PTUX//DOCUMENT Practical Transformation Using XSLT and XPath 2011-02-11 21:00UTC//EN

Fourteenth Edition - 2011-02-11
ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath (XSL Transformations and the XML Path Language)

Crane Softwrights Ltd.
http://www.CraneSoftwrights.com

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath (Prelude) (cont.)

Preface

The main content of this book is in an unconventional style primarily in bulleted form

- derivations of the book are used for instructor-led training, requiring the succinct presentation
  - note the exercises included in instructor-led training sessions are not included in the book
- derivations of the book can be licensed and branded for customer use in delivering training
- the objective of this style is to convey the essence and details desired in a compact, easily perused form, thereby reducing the search for key words and phrases in lengthy paragraphs
- each chapter of the book corresponds to a module of the training
- each page of the book corresponds to a frame presented in the training
- a summary of subsections and their pages is at the back of the book

Much of the content is hyperlinked both internally and externally to the book in the 1-up full-page sized electronic renditions:

- (note the Acrobat Reader "back" keystroke sequence is "Ctrl-Left")
- page references (e.g.: Chapter 2 Getting started with XSLT and XPath (page 46))
- external references (e.g.: `http://www.w3.org/TR/1999/REC-xslt-19991116`)
- chapter references in book summary
- section references in chapter summary
- subsection references in table of contents at the back of the book
- hyperlinks are not present in the cut, stacked, half-page, or 2-up renditions of the material

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Series: Practical Transformation Using XSLT and XPath

Reference:

Pre-requisites:

- knowledge of XML syntax
- knowledge of HTML

Outcomes:

- awareness of documentation
- introduction to objectives and purpose
- exposure to example scripts
- understanding of processing model and data model
- basic script and module writing for transformation
- an overview of every element in the recommendations
- an overview of every function in the recommendations
- introduction to XSL formatting semantics

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Transforming structured information

This book is oriented to the stylesheet writer, not the processor implementer

- certain behaviors important to an implementer are not included
- objective to help a stylesheet writer understand the language facilities needed to solve their problem
  - a language reference arranged thematically to assist comprehension
  - a different arrangement than the Recommendations themselves

This book covers every element, every attribute and every function of both XSLT and XPath, both versions 1.0 and 2.0:

- P1 content specific to XPath 1.0 is marked with a "P1" icon at the beginning of the line
- P2 content specific to XPath 2.0 is marked with a "P2" icon at the beginning of the line
- T1 content specific to XSLT 1.0 is marked with a "T1" icon at the beginning of the line
- T2 content specific to XSLT 2.0 is marked with a "T2" icon at the beginning of the line

First two chapters are introductory in nature

- overview of context of XSLT and XPath amongst other members of the XML family of Recommendations
- basic flow diagrams illustrate use of XSLT
- basic terminology and approaches are defined and explained

Third and fourth chapters cover essential bases of understanding

- data model and processing model for document representation and behavior
- important to understand the models in order to apply the language features

Fifth through ninth chapters address XSLT vocabulary

- every element, attribute and function not already covered when describing the models
- no particular order of the chapters, but example code only uses constructs already introduced in earlier content

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Transforming structured information (cont.)

First two annexes overview HTML and XSL-FO as related to using XSLT

- considerations of using XSLT features to address basic result vocabulary requirements

Third annex includes a number of handy summaries derived from the Recommendations

- alphabetical lists of elements and functions
- print-oriented summaries of all productions

Last annex addresses questions regarding tools

- lists of questions for processor implementers when assessing tool capabilities

External ZIP file included with the purchase of the book

- all of the complete scripts utilized in the documentation as stand-alone files ready for analysis and/or modification
- sample invocation scripts for Windows environments

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 1 - The context of XSLT and XPath

- Introduction - Overview
- Section 1 - The XML family of Recommendations
- Section 2 - Transformation data flows

Outcomes:

- an understanding of the roles of and relationships between the members of the XML family of Recommendations (related to XSLT and XPath)
- an awareness of available documentation and a small subset of publicly available resources
- an understanding of the data flows possible when using XSLT in different contexts and scenarios

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Overview

This chapter reviews the roles of the following Recommendations in the XML family and overviews contexts in which XSLT and XPath are used.

Extensible Markup Language (XML)

- hierarchically describes an instance of information
  - using embedded markup according to rules specified in the Recommendation
  - information is identified with a vocabulary of labels (a set of element types each with a name, a structure and optionally some attributes) described by the user
- optionally specifies a mechanism for the formal definition of a vocabulary
  - controls the instantiation of new information
  - validates existing information is using the expected set of labels

XML Path Language (XPath)

- the document model and addressing basis for XSLT and XQuery

Extensible Stylesheet Language Family (XSLT/XSL/XSL-FO)

- XSL Transformations (XSLT)
  - specifies the transformation of structured information into a hierarchy using the same or a different document model *primarily for the kinds of transformations for use with XSL*
- XSL (Formatting Semantics, a.k.a. XSL-FO)
  - specifies the vocabulary and semantics of the formatting of information for paginated presentation
  - colloquially referred to at times as XSL Formatting Objects

Namespaces

- disambiguates vocabularies when mixing information from different sources
- identifies the dictionary for the labels used to mark up information

Stylesheet Association

- names resources as candidates to be utilized as a stylesheet for processing an XML document
  - does not modify the structural markup of the data
  - used to specify the rendering of an instance of information

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Markup Language (XML)

- `http://www.w3.org/TR/REC-xml`
- `http://www.w3.org/TR/xml11`

A Recommendation fulfilling two objectives for information representation:

- expressing information in a hierarchical arrangement using XML-defined markup
- restricting and/or validating the use of XML markup according to user-specified constraints

Document description and data description

- the roots of XML are from the ISO specification for Standard Generalized Markup Language (SGML) used for document description
- any hierarchical arrangement can be expressed using XML
- any non-hierarchical arrangement can be expressed hierarchically using XML
- XML now commonly used for the description of many kinds of data because of the platform independence of the use of markup and Unicode text

XML defines basic constraints on physical and logical hierarchies of syntax

- the concept of well-formedness with a syntax for markup languages
  - the vocabulary and hierarchy of constructs in an instance of information is ***implicit*** according to the specified rules governing syntactic structures
- a language for specifying how a system can constrain the allowed logical hierarchy of information structures
- the semantics of the user's vocabulary are not formally defined using XML constructs
  - can be described in XML comments using natural language
  - are defined by the applications acting on the information

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Markup Language (XML) (cont.)

Physical hierarchy (the content organization):

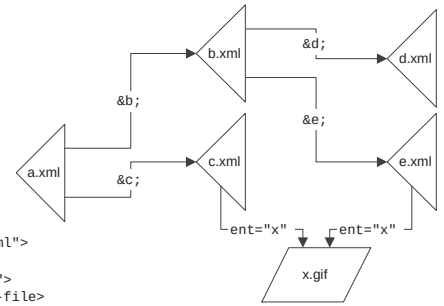
- single collection of information ("XML instance") from multiple physical resources ("XML entities")
  - an XML file is not required to be comprised of more than one physical entity
  - physical modularization typically used to manage a large information set in smaller fragments
  - inappropriately used for XML fragment sharing due to parsing context
- resource is nested syntactically using XML external parsed general entity construct
  - each physical resource has a well-formed logical hierarchy
- unparsed data entities in a declared notation are outside of the parsed hierarchy

Files:

```
adir/a.xml
adir/c.xml
adir/x.gif
bdir/b.xml
bdir/e.xml
bdir/ddir/d.xml
```

a.xml:

```
<!ENTITY b SYSTEM "../bdir/b.xml">
<!ENTITY c SYSTEM "c.xml">
<!ENTITY d SYSTEM "../bdir/ddir/d.xml">
<!ENTITY e SYSTEM "../bdir/e.xml">
<!NOTATION gif-file SYSTEM "gif-uri">
<!ENTITY x SYSTEM "x.gif" NDATA gif-file>
```



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Markup Language (XML) (cont.)

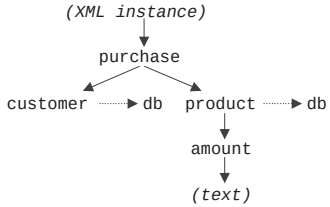
Logical hierarchy (the information):

- single collection of information ("XML instance") comprised of multiple nested containers (XML elements, attributes, text, etc.) where each container is labeled with a name
- each piece is expressed using an XML construct at a user-defined granularity
- the nested breakdown of the information is hierarchical

```
<?xml version="1.0"?>
<purchase>
  <customer db="cust123"/>
  <product db="prod345">
    <amount>23.45</amount>
  </product>
</purchase>
```

The implicit document model exists by the mere presence of logical hierarchy

- the markup of the XML constructs demarcates the locations of the information in the hierarchy
- data model is comprised of family-tree-like relationships of parent, child, sibling, etc.



A logical hierarchy need not come from XML syntax

- through "data projection" the logical tree of any information that can be organized as if it came from XML syntax is indistinguishable from that tree that actually does come from XML syntax
- the data model doesn't retain whatever syntax was used (XML or otherwise) to create the logical tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

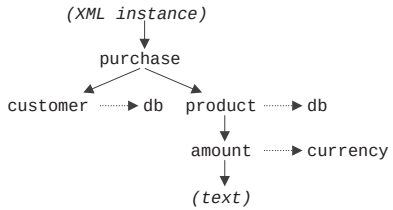
Extensible Markup Language (XML) (cont.)

XML allows user constraints on the logical hierarchy (the vocabulary)

- defines the concept of validity with a syntax for a meta-markup language
- Document Type Definition (DTD) describes the document model as a structural schema
  - the vocabulary defines the logical hierarchy of the information constructs *explicitly* according to user-specified constraints
- other structural and content schema languages exist for XML
  - validation constraints extend to values found within text and attribute content
  - different approaches to describing models provide different benefits
- constrains during generation and confirms during processing
- does not convey semantics of information being marked up

```
<?xml version="1.0"?>
<!DOCTYPE purchase [
<!ELEMENT purchase ( customer, product+ )>
<!ELEMENT customer EMPTY>
<!ATTLIST customer db CDATA #REQUIRED>
<!ELEMENT product  ( amount )>
<!ATTLIST product  db CDATA #REQUIRED>
<!ELEMENT amount   ( #PCDATA )>
<!ATTLIST amount   currency ( GBP | CAD | USD ) "USD"> ]>
<purchase>
  <customer db="cust123"/>
  <product db="prod345">
    <amount>23.45</amount>
  </product>
</purchase>
```

The DTD can supplement the data model with additional information:



- note how the shape of the tree is different in the presence of defaulted attribute declarations
  - the currency attribute is included in the tree when the DTD is present
  - without the DTD the logical tree for the sample instance does not include the currency attribute
  - the markup used is identical in both the example instances

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Markup Language (XML) (cont.)

The equivalent set of document constraints on the logical hierarchy expressed using W3C Schema could be in `purc.xsd`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="purchase">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="customer">
        <xsd:complexType>
          <xsd:attribute name="db" use="required"/>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="product" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="amount">
              <xsd:complexType mixed="true">
                <xsd:attribute name="currency" default="USD">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                      <xsd:enumeration value="GBP"/>
                      <xsd:enumeration value="CAD"/>
                      <xsd:enumeration value="USD"/>
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:attribute>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
          <xsd:attribute name="db" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Markup Language (XML) (cont.)

The hint that a particular W3C Schema applies to a document is given via reserved attributes

- a processor is not obliged to use the hints

```
<?xml version="1.0"?>
<purchase xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="purc.xsd">
  <customer db="cust123"/>
  <product db="prod345">
    <amount>23.45</amount>
  </product>
</purchase>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Extensible Markup Language (XML) (cont.)

DTD declarations affecting the information set of the instance are significant to transform processing that is focused on the implicit logical model of the instance:

- some attribute declarations in DTD are significant
  - attribute list declarations impact transform processing by modifying the information set of the instance
    - supply of defaulted attribute values for attributes not specified in start tags and empty tags of elements
    - declaration of `ID`-typed attributes (for `ID`/`IDREF` processing) that confer element identification uniqueness in an instance
    - declaration of attribute types affecting the attribute value normalization during XML processing
    - attribute information does not affect the well-formed nature of an XML instance
- all DTD content model declarations are not significant
  - what the logical model could contain does not affect what the actual logical model does contain

P2 W3C Schema declarations inform the construction of the data model for the XML instance from the Post Schema Validation Infoset

- only when a schema-aware processor is being used ***and*** when validation is engaged for the source files
  - schema type assignment, default attribute and element value provision, white space normalization of element content
  - the user-supplied lexical form of elements and attributes with atomic schema types may be lost
- when not validated, input information items are treated per the XML information set
  - considered as having unknown data types
- DTD default attribute value declarations override W3C Schema defaults

P1 No respect of element content white space is implied by the content models

- a content model is defined as either element content (a content model without `#PCDATA`) or mixed content (a content model with `#PCDATA`)
- the term "element content white space" is defined in `http://www.w3.org/TR/xml-infoset`
  - sometimes colloquially termed elsewhere as "ignorable white space"
- ***all*** white space is significant to most XSLT 1 processors
- some recognition of white space can be influenced by the XSLT stylesheet

P2 White space text node disposition is at user request

- strip all, preserve all, strip ignorable

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Extensible Markup Language (XML) (cont.)

Chapter 1 - The context of XSLT and XPath
Section 1 - The XML family of Recommendations

XML Recommendation describes behavior

- required of an XML processor
- how it must process an XML stream and identify constituent data
- the information it must provide to an application
- note that programming interfaces that have been standardized are separate initiatives and are ***not*** defined by the XML Recommendation
  - tree-oriented paradigm using DOM (Document Object Model)
  - stream-oriented paradigm using SAX (Simple API for XML)

An XML document is only a labeled hierarchy of information

- XML only unambiguously identifies constituent parts of a stream of hierarchical information
- no inherent meanings or semantics of any kind associated with element types

No rendition or transformation concepts or constructs

- information representation only, not information presentation or processing
- no defined controls for implying rendering semantics
- the `xml:space` attribute signals whether white space in content is significant to the data definition

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XML information links

Links to useful information

- `http://www.xml.com/axml/axml.html` - annotated version of XML 1.0
- `http://xml.coverpages.org/xml.html` - Robin Cover's famous resource collection
- `http://xml.coverpages.org/xll.html` - Extensible Linking Language
- `http://xml.silmaril.ie/` - Peter Flynn FAQ
- `http://www.xmlbooks.com/` - a summary of available printed books
- `http://www.CraneSoftwrights.com/links/trn-20110211.htm` - training material
- `http://www.CraneSoftwrights.com/resources` - free resources
- `http://XMLGuild.info` - consulting and training expertise
- `http://wiki.eclipse.org/PsychoPathXPathProcessor` - standalone XPath 2.0 processor
- `http://xml.coverpages.org/elementsAndAttrs.html` - a summary of opinions
- `http://google-styleguide.googlecode.com/svn/trunk/xmlstyle.html` - a corporate perspective

Related initiatives and specifications

- `http://www.w3.org/TR/2004/REC-xml-infoset-20040204` - XML Information Set
- `http://www.w3.org/TR/xmlschema-0/` - W3C XML Schema
- `http://www.relax-ng.org` - ISO/IEC 19757-2 RELAX NG (based on RELAX and TREX)
- `http://www.schematron.com` - ISO/IEC 19757-3 Schematron
- `http://www.nvdl.org` - ISO/IEC 19757-4 Namespace-based Validation Dispatching Language (NVDL)
- `http://www.w3.org/TR/DOM-Level-2/` - Document Object Model Level 2
- `http://www.saxproject.org` - Simple API for XML

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XML Path Language (XPath)

Representing structured information

- P1 `http://www.w3.org/TR/xpath`
- P2 `http://www.w3.org/TR/xpath20`
- a data model for representing the information found in an XML document as an abstract node tree
  - the original markup syntax is not preserved
  - the user constraints on the document model (e.g. DTD content models) are not germane
  - any logical or physical modularization (the use of entities) is not preserved
- a mechanism for addressing information found in the document node tree
  - the address specifies how to traversal the data model of the instance
- a core upon which extended functionality specific to each of XPointer, XSLT and XQuery is added
  - an expression of Boolean, numeric, string and node values as different data types
  - a set of functions working on the values
- P2 annotated with W3C Schema data type information when available
- P2 data model defined for use with XSLT and XQuery:
  - `http://www.w3.org/TR/xpath-datamodel/`

Addressing and finding structured information

- common semantics and syntax for addressing a logical hierarchy
  - document order, a.k.a. parse order, a.k.a. depth first order
- no representation of the physical hierarchy of an XML document
- a compact non-XML syntax
  - for use in languages needing to address information found in an XML document
  - `id('start')//question[@answer='y']`
    - address all question elements whose `answer` attribute is "`y`" that are descendants of the element in the current document whose unique identifier is "`start`"
    - the result is an address of element nodes
  - P2
```
for $each in id('start')//question[@answer='y']
    return if ($each/@weight) then $each/@weight * 100.
                              else 100.
```
    - for all question elements whose `answer` attribute is "y" that are descendants of the element in the current document whose unique identifier is "`start`", return a sequence of numbers where, if that element has a weight attribute return the weight multiplied by 100, otherwise just return 100
    - the result is a sequence of numbers suitable for processing, such as an argument to the `avg()` function

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XML Path Language (XPath) (cont.)

P1 XPath 1.0 is an addressing language and is *not* a query language

- only based on XML 1.0 and Namespaces in XML 1.0
  - expressed in terms of the XML Information Set
  - `http://www.w3.org/TR/xml-infoset`
- only addresses information that needs to be found in an XML document
- other aspects of querying involve working with the information that is addressed before returning a result to the requestor
  - instructions in XSLT perform query functionality
- XPath is used only to address components of an XML instance, and in and of itself does not provide any traditional query capabilities (though hopefully would be considered as the addressing scheme by those defining such capabilities)

P2 XPath 2.0 is very much a query language

- based on W3C Schema XSD 1.0 perspective of an XML document
- supports conditional expressions, actions on the result set, etc.
- very powerful and expressive language for manipulating all types of information before returning the result of manipulation for action

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Styling structured information

Styling is *transforming* and *formatting* information

- the application of two processes to information to create a rendered result
- the ordering of information for creation isn't necessarily (or shouldn't be constrained to) the ordering of information for presentation or other downstream processes
  - it is a common (though misdirected) first step for people working with these technologies to focus on presentation
  - the ordering should be based on business rules and inherent information properties, not on artificial presentation requirements
  - downstream arrangements can be derived from constraints imposed upstream in the process
  - information created richly upstream can be manipulated into less-richly distinguished information downstream, but not easily the other way around
  - exception when the business rules are presentation or appearance oriented (e.g. book publishing)
- the need to present information in more than one arrangement requires transformation
- the need to present information in more than one appearance requires formatting

W3C XSL Working Group

- chartered to define a style specification language that covers at least the formatting functionality of both CSS and DSSSL
- not intended to replace CSS, but to provide functionality beyond that defined by CSS
  - e.g. add element reordering and pagination semantics

Two W3C Recommendations

- designed to work together to fulfill these two objectives
- XSL Transformations (XSLT) - versions 1.0 and 2.0
  - transforming information obtained from a source into a particular reorganization of that information to be used as a result
- Extensible Stylesheet Language (XSL/XSL-FO) - versions 1.0 and 1.1
  - specifying and interpreting formatting semantics for the rendering of paginated information
  - the acronym XSL-FO is unofficial but in wide use, including at the W3C, for just the formatting objects, properties and property values
  - XSL normatively includes XSLT by reference in chapter 2
    - XSLT has specific features designed to be used with XSL-FO

XSLT and XSL-FO are endorsed by members of WSSSL

- an association of researchers and developers passionate about markup technologies

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

- `http://www.w3.org/TR/2001/REC-xsl-20011015/`
- `http://www.w3.org/TR/xsl11` (`http://www.w3.org/TR/xsl`)

Paginated flow and formatting semantics vocabulary

- capturing agreed-upon formatting semantics for rendering information in a paginated form on different types of media
- XSLT is normatively referenced as an integral component of XSL as a language to transform an instance of an arbitrary vocabulary into the XSL-FO XML vocabulary
- XSL-FO can be regarded simply as a "pagination markup language"
- flow semantics from the DSSSL heritage
  - e.g. headers, footers, page numbers, page number citations, columns, etc.
- formatting semantics from the CSS heritage
  - e.g. visual properties (font, color, etc.) and aural properties (speak, volume, etc.)

Target of transformation

- the stylesheet writer transforms a source document into a hierarchy that uses only the formatting vocabulary in the result tree
- stylesheet is responsible for constructing the result tree that expresses the desired rendering of the information found in the source tree
  - the XML document gets transformed into its appearance
- stylesheet cannot use any user constructs as they would not be recognized by an XSL rendering processor
  - for example, the rendering engine doesn't know what an invoice number or customer number is that may be represented in the source XML
  - the rendering engine does know what a block of text is and what properties of the block can be manipulated for appearance's sake
  - the stylesheet transforms the invoice number and customer number into two blocks of text with specified spacing, font metrics, and area geometry

Device-independent formatting constructs

- the XSL-FO vocabulary describes two media interpretations for objects and properties:
  - visual media
  - aural media
  - a further distinction is also made at times for interactive media
- the results of applying a single stylesheet can be rendered on different types of rendering devices, e.g.: print, display, audio, etc.
- may still be appropriate to have separate stylesheets for dissimilar media
  - device independence allows the information to be rendered on different media, but a given rendering may not be conducive to consumption

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Addressing, querying and publishing structured information

- T1 `http://www.w3.org/TR/xslt`
  - addressing structured information
- T2 `http://www.w3.org/TR/xslt20`
  - querying structured information
- a framework for complex and intelligent querying of structured content
  - with a powerful syntax for modular and extensible stylesheet writing
- works on XML documents
- works on any source of information projected as if it were an XML document
  - such projection is defined by the vendor, not by the specification
  - the specification sees all information as if it had been in an XML document
  - e.g. database tables, rows and columns
  - e.g. unstructured documents
  - e.g. proprietary binary formats
  - any information can be fit (or shoehorned) into an XML document by using data projection
- numerous features for publishing information for human consumption
  - e.g. formatting numbers, dates and times
  - e.g. polymorphism of stylesheet constructs for specialization of behaviors
  - e.g. elaborate grouping criteria
  - e.g. multiple result trees

Shares the same data model as XQuery

- built on XPath 2.0 with additional functions not available in XQuery expressions

Shares the same basic processing model as XQuery

- some XSLT and XQuery implementations share the same core engine
  - e.g. Saxon 9 `http://saxon.sf.net` treats XSLT and XQuery merely as different syntax skins over the same implementation engine

Shares the same serialization specification as XQuery

- used to frame query results as structured or non-structured output of transformation

Syntactically, XSLT is an XML vocabulary

- an XSLT stylesheet is a well-formed XML document
- all use of XPath 2.0 is in attributes of XSLT and other XML elements

Transformation specifications are termed "XSLT stylesheets"

- describing how new results are constructed from old inputs
- termed generically as "a transform" in this training material

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Stylesheet Language Transformations (XSLT) (cont.)

Transformation using construction by example

- a vocabulary for specifying templates of the result that are filled-in with information from the source
  - the stylesheet includes examples of each of the components of the result
  - the stylesheet writer declares how the XSLT processor builds the result from the supplied examples
- the primary memory management and manipulation (node traversal and node creation) is handled by the XSLT processor using declarative constructs, in contrast to a transformation programming language or interface (e.g. the DOM - Document Object Model) where the programmer is responsible for handling low-level manipulation using imperative constructs
- includes constructs to reposition over structures and information found in the source
- the information being transformed can be traversed in different ways any number of times required to construct the desired result
- straightforward problems are solved in straightforward ways without needing to know programming
  - useful, commonly-required facilities are implemented by the processor and can be triggered by the stylesheet
  - the language is Turing complete, thus arbitrarily complex algorithms can be implemented (though not necessarily in a pretty fashion)
- includes constructs to manage stylesheets by sharing components in different fragments
- T2 XSLT 2.0 has many more programming features and function calls than XSLT 1.0

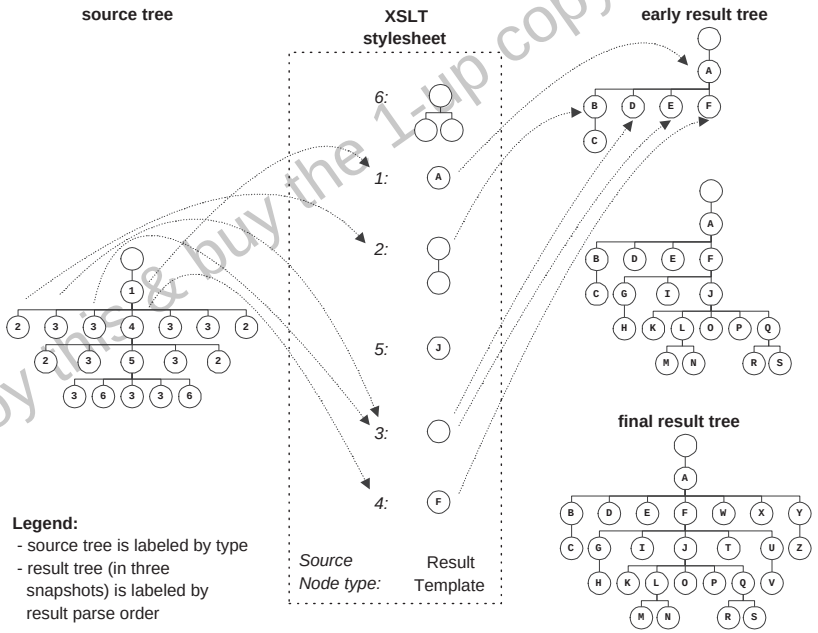
Many language features for modularization and leveraging stylesheets

- supports forms of polymorphism for stylesheet constructs
- supports extensive re-use of stylesheet fragments for generalized transformations or specific transformations
- overriding template rules
  - allows one to create "onion skins" of modifications to stylesheet libraries
- testing the presence of extensions before using them
  - allows one to run one stylesheet with multiple XSLT processors

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible Stylesheet Language Transformations (XSLT) (cont.)

Illustration of templates triggered in source-tree order constructing a result:



Of note:

- the source tree contains nodes of six different types, labeled "1" through "6"
  - a number of nodes are found multiple times in the source tree
- the stylesheet contains fragmented examples of the result tree
  - each example template is associated with a node in the source tree
- the nodes in the source tree trigger the building of the result from the example templates
  - some examples are used multiple times in the result
- in this example, the source tree is visited strictly in parse order to generate the result tree
  - the stylesheet can visit the source tree in whatever order is required to trigger the assembly of the result tree in result parse order
  - result parse order is indicated by the letters "A" through "Z"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XSLT properties

Expression syntax is iconic

- using XML markup allows one to *manifest* the output
  - XML is a first-class data type and output expression syntax
  - the syntax itself is abstracted into a tree of nodes
  - syntax not related to the information in the document is not preserved
- using other languages one must *describe the creation* of the output
  - XML is created using function calls, not built into the language syntax

Abstract structure result of nodes, not markup

- external result markup (if needed) is determined from the result node tree
- the result of transformation is a tree of nodes built from instantiated templates as an internal hierarchy that *may* be serialized externally as markup
- the processor may, but is not obliged to, externalize the result tree in XML or some other type of syntax if requested by the transform writer
  - the transform writer has little or no control over the syntactic constructs chosen by the processor for serialization
  - the transform writer can request certain behaviors that the processor can ignore
  - final result is guaranteed to comply with lexical requirements of the output method
    - when not coerced by certain transform controls
  - source tree markup syntax preservation cannot be implemented with a transform
    - because the source tree syntax is translated into source tree nodes and forgotten
- the processing model allows the processor to immediately serialize the result tree as markup while it is being built by the transform, and not maintain the complete result in memory
- the transform may request the processor emit the result tree using built-in available lexical conventions (XML, HTML or text-only conventions)
- T2 multiple result trees may be constructed and serialized

Not intended for syntactic general purpose XML transformations

- designed for downstream-processing and subsequent transformations or interpretation
  - does not include certain features appropriate for syntax-level general purpose transformations
    - unsuitable for original markup syntax preservation requirements
  - T2 XSLT 2.0 has more syntax serialization features than XSLT 1.0
  - includes facilities for working with the XSL vocabulary easily
- still powerful enough for *most* downstream-processing transformation needs
  - where the syntax choices when using XML are not important
  - absolutely general purpose when the output is going to be input to an XML processor

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT properties (cont.)

Document model and vocabulary independent

- a transform is independent of any Document Type Definition (DTD) or schema that may have been used to constrain the instance being processed
- a processor can process well-formed XML documents without a model
  - behavior is specified against the presence of markup in an instance as the implicit model, not against the allowed markup prescribed by any explicit model
- one transform can process instances of different document models
- multiple instances of different models can be used in a single transformation
- different transforms can process a given single instance to produce different results

Source files and transforms

- one or more source files and one or more transform fragments
  - starting with a single source file and the top-most transform fragment
- T1 all stylesheets and source files must be well-formed XML
- T2 stylesheets must be XML, source files may be simple text or well-formed XML
  - zero or more source files and one or more stylesheet fragments
  - starting with the top-most stylesheet fragment and optionally a source file
- the processor is allowed to deliver well-formed XML from any data source
- Recommendation does not support SGML instances as input
  - see `http://www.w3.org/TR/NOTE-sgml-xml-971215` for a comparison of SGML and XML
  - see `http://tidy.sourceforge.net/` for interpretation and conversion of instances of the HTML vocabulary into XHTML markup conventions
  - see `http://www.ccil.org/~cowan/XML/tagsoup` for interpretation and conversion of streams of arbitrary HTML constructs
  - see `http://www.jclark.com/sp/sx.htm` in the SP package `http://www.jclark.com/sp` for conversion of SGML instances to XML instances without document type declarations
  - see `http://www.CraneSoftwrights.com/resources/n2x` for conversion of SGML instances to XML instances with document type declarations

Validation unnecessary (but convenient)

- an XSLT processor need not implement a validating XML processor
- must implement at least a non-validating XML processor to ensure well-formedness
- validation is convenient when debugging transform development
  - if the source document does not validate to the model expected by the transform writer, then a correctly functioning transform may exhibit incorrect behavior
  - time spent debugging the working transform is wasted if the source is incorrect
- can selectively validate input documents and result documents using W3C Schema

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XSLT properties (cont.)

Multiple source files possible

- 1 one mandatory primary source file
- one optional primary source file
  - in the absence of a source file a named template must be specified as where to start processing
- transform may access arbitrary other source files
  - including itself as a source file
  - names of resources hardwired within the transform
  - names of resources found within source files
- multiple accesses to the same resource refer to a single abstract representation
  - one is not built for each access to a named resource
- 2 simple text files can be input into the process

Extensible language design supplements processing

- a processor *may* support extensions specified in the transform but is not obliged to do so
  - extended functions
  - extended serialization conventions
  - extended sorting schemes
  - extended instructions
- access to non-standardized extensions is specified in standardized ways
- transform user-defined functions can be declared and used

Single-pass construction of the result node-tree

- unlike the Document Object Model (DOM)
  - reified node-tree manipulation (read/write) interface with syntax serialization
- unlike the Simple API for XML (SAX)
  - single-pass input event-handling interface with single-pass result markup syntax
- transform must construct the result tree in result-tree parse order *in one pass*
  - no revisiting of the result tree after construction
  - no revisiting an element's start tag after beginning that element's content
  - recall the result tree building shown on page 23
- the source trees can be traversed in any order (not necessarily in parse order)
  - information in the source trees can be ignored or selectively processed
- the result tree is emitted as if constructed chronologically in parse order
  - this is not an implementation constraint, but an implementation must act as if the tree were created in parse order
    - an important distinction for parallelism where partial trees may be constructed in parallel

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Historical development of the XSL and XQuery Recommendations

Recommendation release history:

- first concept description floated in August 1997 with no official status within the World Wide Web Consortium (W3C)
  - `http://www.w3.org/TR/NOTE-XSL.html`
- the XSL Working Group officially chartered in early 1998
  - `http://www.w3.org/Style/XSL/`
- agreed upon requirements for XSL by the Working Group:
  - `http://www.w3.org/TR/WD-XSLReq`
- the XSL 1.0 Recommendation (XSL-FO) published October 15, 2001
  - `http://www.w3.org/TR/2001/REC-xsl-20011015/`
- the XSL 1.1 Recommendation (XSL-FO) published December 5, 2006
  - `http://www.w3.org/TR/2006/REC-xsl11-20061205/`
- the XSLT/XPath 1.0 Recommendations published November 16, 1999
  - `http://www.w3.org/TR/1999/REC-xslt-19991116`
    - `http://www.w3.org/1999/11/REC-xslt-19991116-errata` - errata
  - `http://www.w3.org/TR/1999/REC-xpath-19991116`
    - `http://www.w3.org/1999/11/REC-xpath-19991116-errata` - errata
- XSLT 1.1 (work abandoned)
  - `http://www.w3.org/TR/2000/WD-xslt11req-20000825` - requirements
  - `http://www.w3.org/TR/2001/WD-xslt11-20010824`
  - no incompatible changes to XSLT 1.0 in XSLT 1.1, only additional functionality
  - too many interactions with plans for XSLT 2.0, so functionality to be folded into XSLT 2.0 release
- XSLT 2.0/XPath 2.0/XQuery 1.0 originally published January 23, 2007, followed by editorial editions:
  - `http://www.w3.org/TR/2007/REC-xslt20-20070123/`
  - `http://www.w3.org/TR/2010/REC-xpath20-20101214/`
  - `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`
  - `http://www.w3.org/TR/2010/REC-xpath-functions-20101214/`
  - `http://www.w3.org/TR/2010/REC-xslt-xquery-serialization-20101214/`
  - `http://www.w3.org/TR/2010/REC-xquery-20101214/`
  - `http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/`
  - `http://www.w3.org/TR/2010/REC-xqueryx-20101214/`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSL information links

Links to useful information

- `http://xml.coverpages.org/xsl.html` - Robin Cover
- `http://www.mulberrytech.com/xsl/xsl-list/` - mail list
- `http://www.dpawson.co.uk` - an XSL/XSLT FAQ
- `http://www.zvon.org/HTMLonly/XSLTutorial/Books/Book1/index.html` - numerous example XSLT scripts and fragments
- `http://www.openmath.org/cocoon/openmath/` - OpenMath project work by David Carlisle
- `http://www.CraneSoftwrights.com/links/trn-20110211.htm` - comprehensive XSLT/XPath and XSL-FO training material
- `http://XMLGuild.info` - consulting and training expertise
- `http://www.CraneSoftwrights.com/resources`- free XSLT and XSL-FO resources
- `http://incrementaldevelopment.com/xsltrick/` - "Stupid XSLT Tricks"
- `http://xml.coverpages.org/xslSoftware.html` - list of tools
- `http://www.exslt.org/` - community effort for XSLT extensions
- `http://exslfo.sf.net` - community effort for XSL-FO extensions
- `http://foa.sourceforge.net/` - open source FO GUI authoring tool
- `http://www.xslfast.com/` - commercial FO GUI authoring tool
- `http://www.inventivedesigners.com/` - commercial FO GUI authoring tool
- `http://www.abisource.com/` - word processing with "Save As..." for XSL-FO
- `http://www.AntennaHouse.com/XSLsample/XSLsample.htm` - paginating XHTML
- `ISBN 1-56609-159-4` - "The Non-Designer's Design Book", Robin Williams, Peachpit Press, Inc., 1994
- `ISBN 0-8230-2121-1/0-8230-2122-X` - "Graphic design for the electronic age: The manual for traditional and desktop publishing", Jan V. White, Xerox Press, 1988 (out of print but worthwhile to search for as a used book)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespaces

- `http://www.w3.org/TR/REC-xml-names`

An important role in information representation:

- vocabulary distinction in a single XML document
  - mixing information from different document models
  - labels in the hierarchy are globally unique and identifiable
  - a metaphor is that each namespace is a dictionary with words
    - each dictionary may have a different definition for the same word as found in other dictionaries
    - the namespace identifies which dictionary of words is in use
- possible use for resource discovery being considered
  - generalized associated information regarding information in an instance
  - possible access to document model, transforms, validation algorithms, access libraries, etc.

Vocabulary distinction

- specifies a simple method for qualifying element and attribute names used in XML documents
- allows the same element type name to be used from different vocabularies in a given document
  - consider two vocabularies each defining the element type named "`<set>`", each with very different semantics
    - following the metaphor, the one word has two different definitions and interpretations, one from each dictionary
    - in SVG (Scalable Vector Graphics) the element `<set>` refers to setting a value within the scope of contained markup
    - in MathML (Mathematical Markup Language) `<set>` refers to a collection of constructs treated as a set
  - any document needing to mix elements from the two vocabularies may need to use the same name
    - without namespaces an application cannot distinguish which construct is being used
  - a namespace prefix differentiates the element type name suffix in an instance
    - `<svg:set>`
    - `<math:set>`
  - composite name lexically parses as an XML name
    - the use of the colon is defined by the namespaces recommendation
- also used to uniquely distinguish identification labels in some Recommendations
  - e.g.: customized sort scheme label

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Namespaces (cont.)

URI value association

- associates element type name prefixes with Universal Resource Identifier (URI) references *whether or not any kind of resource exists at the URI*
  - following the metaphor, the URI uniquely identifies the dictionary of the words
    - supplemental documentation defines the meaning of each of the words
  - URI domain ownership under auspices of established organization
  - URI conflicts avoided if rules followed
- examples:
  - `xmlns:svg="http://www.w3.org/2000/svg-20000629"`
  - `xmlns:math="http://www.w3.org/1998/Math/MathML"`
  - `xmlns:ex1="urn:isbn:978-1-894049:example"`
  - `xmlns:ex2="urn:X-Crane:namespaces:documents:example2"`
  - `xmlns:ex3="ftp://ftp.CraneSoftwrights.com/ns/example3"`
  - `xmlns:ex4="mailto:gkholman@CraneSoftwrights.com"`
- explicitly does not require to de-reference any kind of information from the URI
  - note that the Resource Description Framework (RDF) recommendation does have a convention of looking to the URI for information, though this is outside the scope of the Namespaces recommendation
- according to the recommendation, the URI is *only* used to disambiguate otherwise identical unqualified members of different vocabularies

The choice of the prefix is arbitrary and can be any lexically valid name

- the prefix is never a mandatory aspect of any Recommendation
- the prefix is discarded by the XML namespace-aware processor along the lines of:
  - `<{http://www.w3.org/2000/svg-20000629}set>`
  - `<{http://www.w3.org/1998/Math/MathML}set>`
  - the above use of "`{`" and "`}`" are a common convention but not standard
  - note how the "`/`" characters of the URI would be unacceptable given the lexical rules of names, thus, the URI could never be used directly in the XML tags
- the prefix is a syntactic shortcut preventing the need to specify long distinguishing strings

Different views of the name of `<svg:set>`:

- "`set`" is the local name
- "`svg:set`" is the qualified name
  - a name subject to namespace interpretation (prefixed or un-prefixed)
  - the lexical space for the W3C Schema QName data type
- "`{http://www.w3.org/2000/svg-20000629}set`" is the expanded name
  - combination of namespace URI (also called "namespace name") and the local part
  - the value space for the W3C Schema QName data type
  - the use of "`{`" and "`}`" is not standard, but is used by some tools such as Saxon
- "`http://www.w3.org/2000/svg-20000629#set`" is a URI value convention

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespaces (cont.)

An example of using namespaces in a Universal Business Language (UBL) invoice:

- for space reasons the lengthy namespace URI strings have been abbreviated
- note that namespaces are important because there are two elements with the same local name "`Location`", one in each of two different namespaces

```
<Invoice xmlns="urn:oasis:...:xsd:Invoice-2"
         xmlns:cbc="urn:oasis:...:xsd:CommonBasicComponents-2"
         xmlns:cac="urn:oasis:...:xsd:CommonAggregateComponents-2"
         xmlns:ext="urn:oasis:...:xsd:CommonExtensionComponents-2"
         xmlns:demo="urn:x-Demo:Demo">
  <ext:UBLExtensions>
    <ext:UBLExtension>
      <cbc:ID>Demo1</cbc:ID>
      <cbc:Name>Demonstration</cbc:Name>
      <ext:ExtensionAgencyID>CSL</ext:ExtensionAgencyID>
      <ext:ExtensionAgencyName>Crane Softwrights Ltd.
</ext:ExtensionAgencyName>
      <ext:ExtensionVersionID>0.1</ext:ExtensionVersionID>
      <ext:ExtensionAgencyURI>http://www.CraneSoftwrights.com/
links/res-dev.htm</ext:ExtensionAgencyURI>
      <ext:ExtensionURI>urn:x-Demo:Demo:0.1</ext:ExtensionURI>
      <ext:ExtensionReasonCode listURI="urn:x-Demo:Demo:ReasonCodes">1
</ext:ExtensionReasonCode>
      <ext:ExtensionReason>Illustration</ext:ExtensionReason>
      <ext:ExtensionContent>
        <demo:Demo>
          <demo:Thing>This is a test</demo:Thing>
          <cbc:ID>DemoTest</cbc:ID>
          <demo:Total currencyID="GBP">100.00</demo:Total>
        </demo:Demo>
      </ext:ExtensionContent>
    </ext:UBLExtension>
  </ext:UBLExtensions>

  <cbc:ID>A00095678</cbc:ID>
  <cbc:IssueDate>2005-06-21</cbc:IssueDate>
  <cbc:Note>sample</cbc:Note>
  <cac:AccountingSupplierParty>
    <cac:Party>
      <cac:PartyName>
...
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespaces (cont.)

Namespaces in XSLT and XSL-FO

- both files are in well-formed XML syntax
  - require all namespaces used to be declared; there are no defaults
- recommendations utilize namespaces to distinguish the desired result tree vocabularies from the transformation instruction vocabularies
- `http://www.w3.org/1999/XSL/Transform`
  - XSL transformation instruction vocabulary
  - the use of any archaic URI values for the vocabulary will not be recognized by an XSLT processor
- `http://www.w3.org/1999/XSL/Format`
  - XSL formatting result vocabulary
  - the year represents when the W3C allocated the URI to the working group, not the version of XSL the URI represents

Extension identification

- processors are allowed to recognize other namespaces in order to implement extensions not defined by the Recommendations:
  - functions
  - XSLT instructions
  - XSLT system properties
  - collations
  - serialization methods
- e.g.: `http://www.jclark.com/xt`
  - extensions available when using XT
- e.g.: `http://saxon.sf.net/`
  - extensions available when using Saxon

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespaces (cont.)

Naming of top-level constructs in XSLT

- libraries of transform fragments can isolate their constructs by using unique namespace URI strings
- building upon an existing library is done without risking the integrity of the existing stylesheets when one is disciplined about the naming of constructs
- in the following example, two different variables are declared because of the unique namespace URI strings (the prefixes are immaterial)
  - the first is in namespace "`urn:X-a`" and the second is in namespace "`urn:X-b`"
- ```
  01 <xsl:variable name="a:thing" select="'abc'" xmlns:a="urn:X-a"/>
  02 <xsl:variable name="a:thing" select="'def'" xmlns:a="urn:X-b"/>
  ```
- T2 stylesheet-defined function names must be namespace qualified
- the default namespace is never used for naming top-level constructs

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.
  ```

Stylesheet association

- `http://www.w3.org/TR/xml-stylesheet`

Relating documents to stylesheets

- associating one or more stylesheets with a given XML document
- same pseudo-attributes and semantics as in the HTML 4.0 recommendation elements:
  - `<LINK REL="stylesheet">`
  - `<LINK REL="alternate stylesheet">`

Ancillary markup

- not part of the structural markup of an instance, thus it is marked up using a processing instruction rather than first-class (declared or declarable in a document model) markup

Typical examples of use:

```
01 <?xml-stylesheet type="text/xsl" href="../xs/xslstyle-docbook.xsl"?>
```

```
01 <?xml-stylesheet type="text/css" href="normal.css"?>
```

Less typical examples provided for by the design:

```
01 <?xml-stylesheet alternate="yes" title="small"
02                  href="small.xsl" type="application/xslt+xml"?>
```

- provide the processor with an alternate stylesheet if some external stimulus triggers it by name

```
01 <?xml-stylesheet href="#style1" type="application/xslt+xml"?>
```

- instruct the processor to find the stylesheet embedded in the source document at the named location

Important note about `type=` values for associating XSLT:

- `type="text/xsl"` is not a registered MIME type
  - the only type recognized by IE for the use of XSLT
- `type="application/xslt+xml"` has been proposed in IETF RFC 3023
- `type="text/xml"` is reported to be supported by some processors

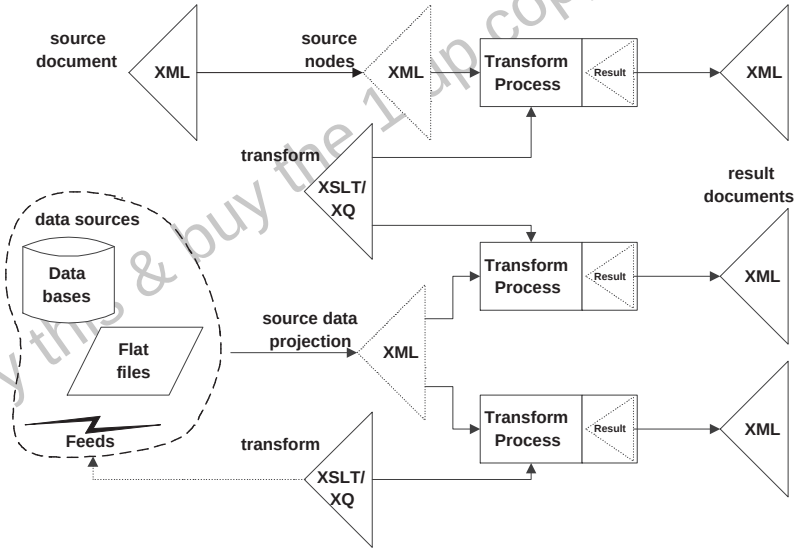
See XSLStyle™ (page 525) for an embedded XSLT documentation methodology

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Transformation from XML to XML

The basic behavior is to transform a hierarchical input into a hierarchical result tree:

- that result tree may be emitted as an XML instance



Of note:

- a given transform can be applied to more than one XML structure
- a given XML structure can have more than one transform applied
- a given XML structure can be derived from an XML file or projected from some other data source identified by the transform
- the result of construction is the abstract result tree within the transform process serialized to the emitted XML under the control of the process
- the dotted triangle in the process represents the abstract node tree of the result

Diagram legend

- processes represented by rectangles
- hierarchical structures represented by triangles
  - a tree structure with the single root at the left point and the tree expanding and getting larger towards the leaves at the right edge
  - XML files are drawn with a solid line, node structures are drawn with a dotted line
- unstructured files represented by parallelograms

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Transformation from XML to non-XML

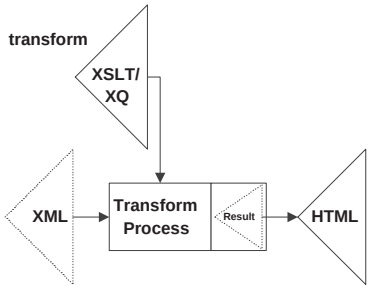
A processor may choose to recognize the transform's request to serialize a non-XML representation of the result tree:

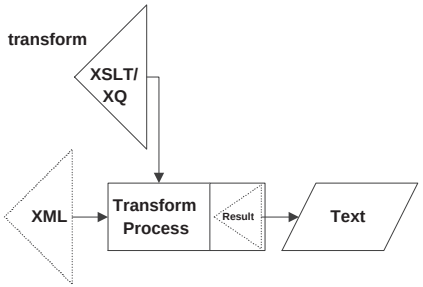
- triggered through using an output serialization method supported by the processor

Shared serialization specification between XSLT 2.0 and XQuery 1.0

- `http://www.w3.org/TR/xslt-xquery-serialization/`

At least two non-XML tree serialization methods common to all specifications:





- `html`
  - HTML markup and structural conventions
    - some older HTML user agents (e.g. browsers) will not correctly recognize elements in the HTML vocabulary when the instance is marked up using XML conventions (e.g. `<br/>` must be `<br>`), thus necessitating the interpretation of HTML semantics when the result tree is emitted
    - using this will not validate the result tree output as being HTML
      - if the result is declared HTML but the desired output isn't HTML, the HTML semantics could interfere with the markup generated
  - HTML built-in character entities (e.g.: accented letters, non-breaking space, etc.)
- `text`
  - simple text content with all element start and end tags removed and ignored
  - none of the characters are escaped on output
  - example of use: creating operating system batch and script files from structured XML documents

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

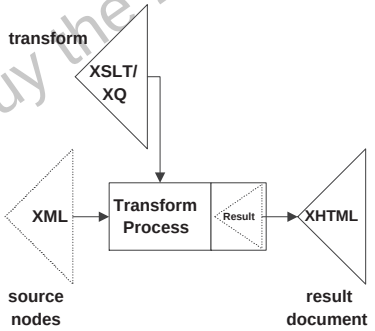
Transformation from XML to non-XML (cont.)

T1 No standardized support for XHTML lexical conventions

- a processor could offer a custom extension, but many (possibly all?) do not

T2 Standardized support for XHTML lexical conventions

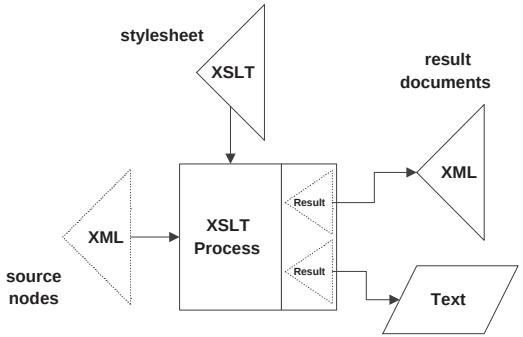
- `xhtml`
  - browser compatibility guidelines for empty tags for elements defined to be empty
  - no markup minimization for empty elements for elements not defined to be empty



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Transformation from XML to non-XML (cont.)

1. Only standardized support for a single result tree
   - most XSLT processors offer a custom extension, but there is no obligation to do so and it is not standardized
2. Standardized support for multiple result trees
   - each result tree can have the same or different serialization
   - multiple result trees are not accessible to a single XSL-FO process

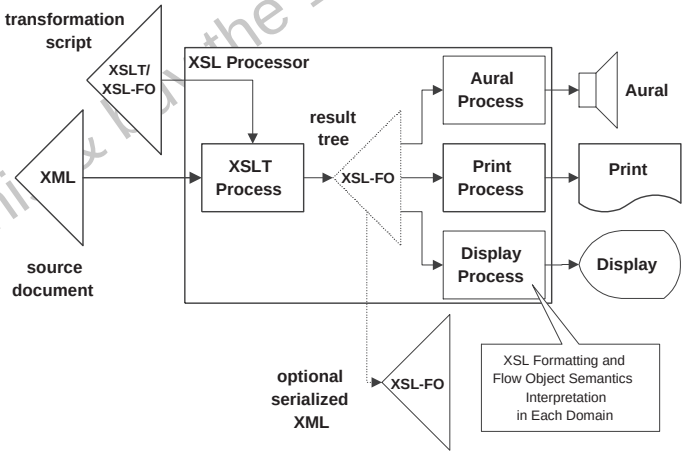


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Transforming and rendering XML information using XSLT and XSL-FO

When the XSLT result tree is specified to utilize the XSL-FO formatting vocabulary:

- the normative behavior is to interpret the result tree according to the formatting semantics defined in XSL for the XSL-FO formatting vocabulary
- an inboard XSLT processor can effect the transformation to an XSL-FO result tree
- the XSL-FO result tree need not be serialized in XML markup to be conforming to the recommendation (though useful for diagnostics to evaluate results of transformation)



Of note:

- the stylesheet contains only the XSLT transformation vocabulary, the XSL formatting vocabulary, and extension transformation or foreign object vocabularies
- the source XML contains the user's vocabularies
- the result of transformation contains exclusively the XSL formatting vocabulary and any extension formatting vocabularies
  - does not contain any constructs of the source XML or XSLT vocabularies
- the rendering processes implement for each medium the common formatting semantics described by the XSL recommendation
  - for example, space specified before blocks of text can be rendered visually as a vertical gap between left-to-right line-oriented paragraphs or aurally as timed silence before vocalized content

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

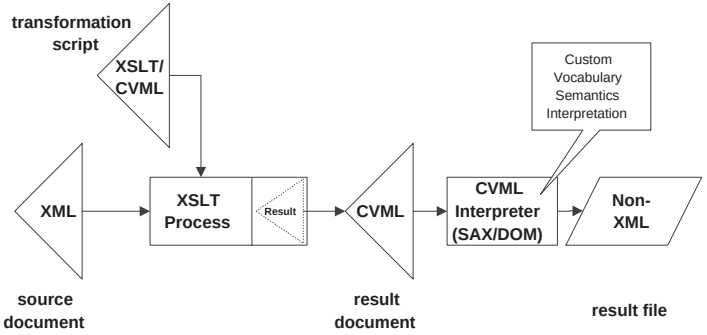
XML to binary or other formats

Some non-XML requirements are neither text nor HTML

- need to produce composition codes for legacy system
- binary files with complex encoding
- custom files with complex or repetitive sequences

One can capture the semantics of the required output format in a custom XML vocabulary

- e.g.: "CVML" for "Custom Vocabulary Markup Language"
- designed specifically to represent meaningful concepts for output



A single translation program (drawn as "CVML Interpreter"):

- can interpret all XML instances using the custom vocabulary markup language (e.g. CVML) to produce the output according to the programmed semantics
- is *independent* of the XSLT stylesheets used to produce the instances of the custom vocabulary
- allows any number of stylesheets to be written without impacting the translation to the final output
- divorces the need to know syntactic output details
  - output is described abstractly by semantics of the vocabulary
  - output is serialized following specific syntactic requirements

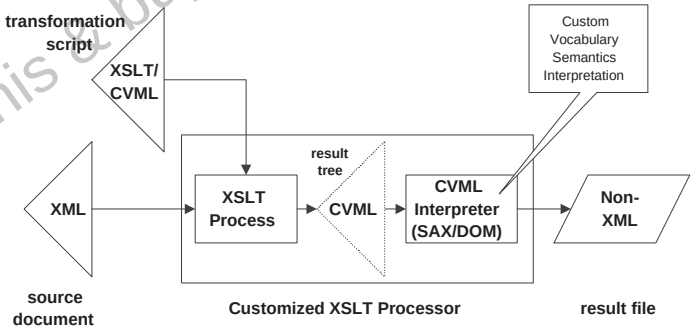
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XML to binary or other formats (cont.)

The XSLT recommendation is extensible providing for vendor-specific or application-specific output methods:

- `xmlns:`*`prefix`*`="`*`processor-recognized-URI`*`"`
- *`prefix`*`:`*`serialization-method-name`*
  - vendors can choose to support additional built-in tree serialization methods
  - output can be textual, binary, dynamic process (e.g.: database load), auditory, or any desired activity or result

The ability to specify vendor-specific or implementation-specific output methods allows custom semantics to be interpreted *within* the modified XSLT processor, thus not requiring the intermediate file:

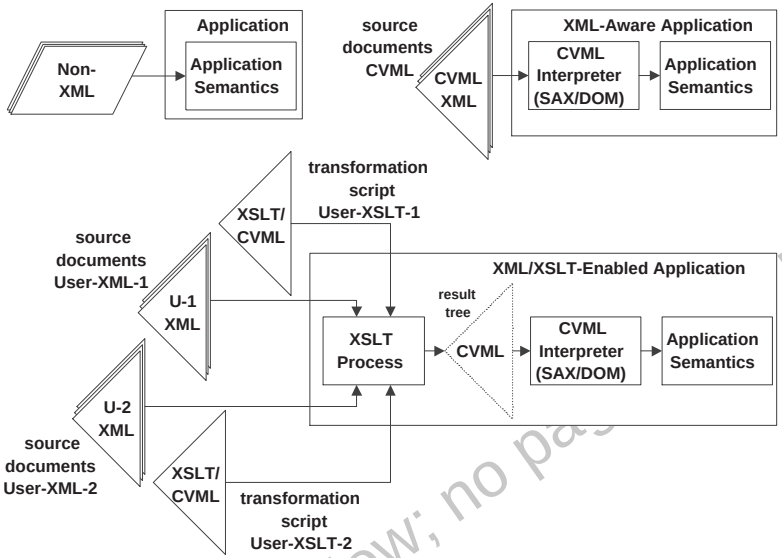


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT as an application front-end

A legacy application can utilize an XSLT processor to accommodate arbitrary XML vocabularies

- making an application XML-aware involves using an XML processor to accommodate a vocabulary expressing application data semantics
  - event driven using SAX processing and programming
  - tree driven using DOM processing and programming
  - without XSLT, each different XML vocabulary would need to be accommodated by different application integration logic
- an application can engage an XSLT processor and directly access the result tree
  - single process programmed to interpret a single markup language
  - each different XML vocabulary is accommodated by only writing a different XSLT stylesheet
  - each stylesheet produces the same application-oriented markup language
- no reification of the result tree is required

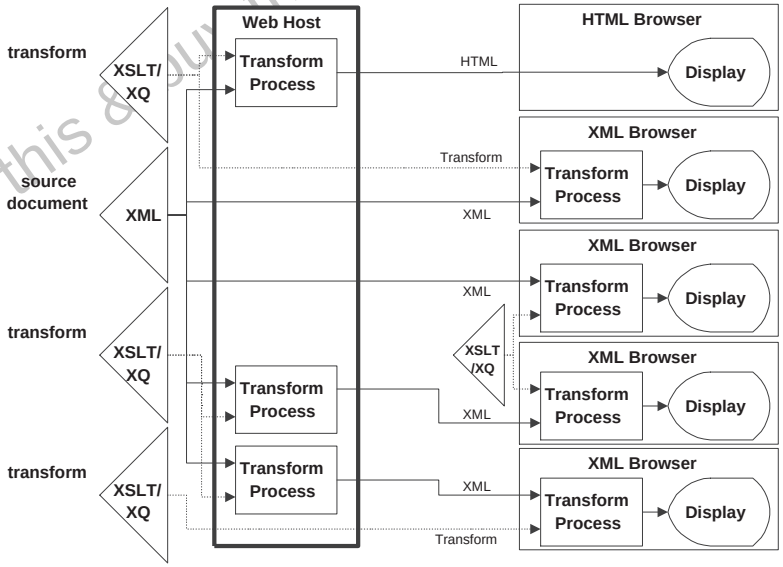


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Three-tiered architectures

To support a legacy of user agents that do not support XML:

- web servers can detect the level of support of user agents
- where XML and XSLT or XQuery are not supported in a user agent:
  - the host can take on the burden of transformation
- where XML and XSLT or XQuery are supported in a user agent
  - the burden of transformation can be distributed to the agent
  - the XML information can be massaged before being sent to the agent
- allows information to be maintained in XML yet still be available to all users



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Three-tiered architectures (cont.)

Always performing server-side transformation:

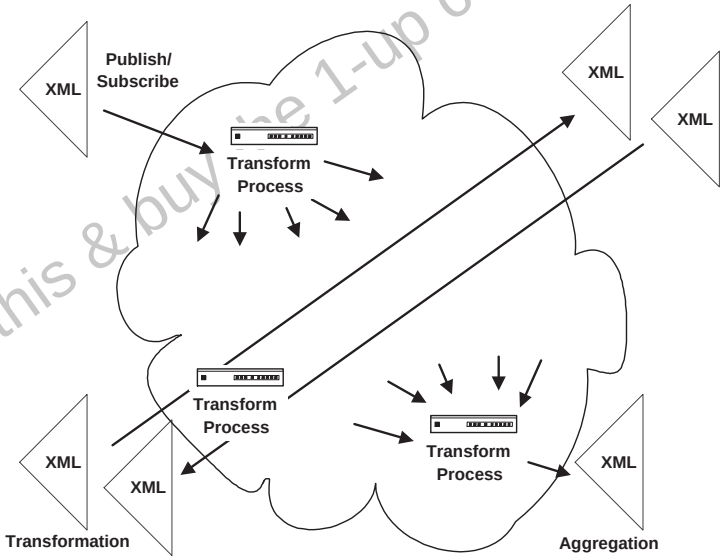
- good business sense in some cases
  - even if technically it is possible to send semantically-rich information
- never send unprocessed semantically-rich XML
  - or only send it to those who are entitled to it
    - for security reasons
    - for payment reasons
- translation into a presentation-orientation
  - using a markup language inherently supported by the user agent (e.g. HTML)
  - using a custom, semantic-less markup language with an associated transformation
- "semantic firewall"
  - to protect the investment in rich markup from being seen where not desired
  - no consensus in the community that semantic firewalls are a "good thing"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT and XQuery on the wire

XSLT and XQuery have a role in a large or small network cloud:

- simple transformation services can be made available to users on the network, unburdening the user's own infrastructure



Publish/Subscribe

- a network service can accept subscription requests from across the network
- the XML document from the publisher is routed to all subscription destinations
- a subscriber can request a transformation process so as to receive the published information in the desired structure

Aggregation

- a network service can accept XML documents from across the network
- a user of the service can receive the aggregate of all of the information
- the information can be transformed into a homogenous collection for ease of processing and analysis

Transformation

- a user of the network can utilize wire-speed transformation of outgoing and incoming documents to a peer

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 2 - Getting started with XSLT and XPath

- Introduction - Getting started
- Section 1 - Transform examples
- Section 2 - Syntax basics
- Section 3 - Approaches to transform design
- Section 4 - More transform examples

Outcomes:

- analyze the different components of a few example transforms
- introduce the concepts of instructions and templates

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Getting started

A few simple transformations:

- using Saxon
  - Saxon 6.5.5 (and later) support XSLT 1.0
  - Saxon 9 (and later) support XSLT 2.0 and XQuery 1.0
- using Internet Explorer 5 or greater
  - for IE5, the updated MSXML processor (at least the third Web Release of March 2000) is needed to support the W3C XSLT 1.0 Recommendation
  - the IE6 production release supports the W3C XSLT 1.0 Recommendation

Dissect example transforms

- identify transform components as an introduction to basic concepts covered in more detail in the later chapters

This material has a number of handy references harvested from the specification documents:

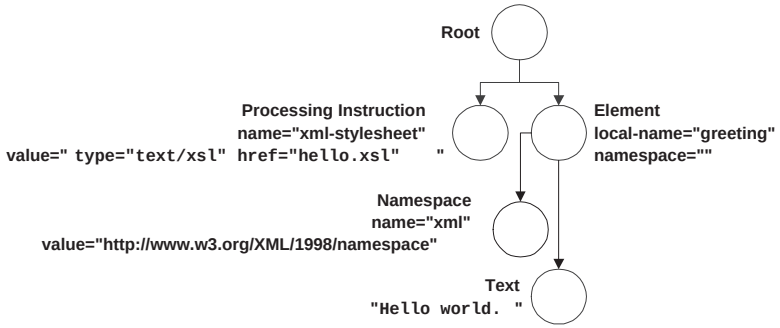
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Some simple examples

Consider the following XML file `hello.xml` obtained from the XML 1.0 Recommendation and modified to declare an associated stylesheet:

```
01 <?xml version="1.0"?>
02 <?xml-stylesheet type="text/xsl" href="hello.xsl"?>
03 <greeting>Hello world.</greeting>
```

This is the complete logical tree of the entire instance:



Note that there is no node in the tree created by the XML Declaration

- the XML Declaration is a syntactic signal in an XML instance regarding the encoding and version of XML being used
- it is consumed by the XML processor as part of the parsing process and is not delivered to an application

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Some simple examples (cont.)

Consider the following XSLT file `hellohtm.xsl` to produce HTML, noting how much it looks like an HTML document yet contains XSLT instructions:

```
01 <?xml version="1.0"?>
02 <!--hellohtm.xsl-->
03 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
04 <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05       xsl:version="1.0">
06  <head><title>Greeting</title></head>
07  <body><p>Words of greeting:<br/>
08    <b><i><u><xsl:value-of select="greeting"/></u></i></b>
09    </p>
10  </body>
11 </html>
12
```

Using an MSDOS command line invocation to execute the stand-alone processor explicitly with a supplied stylesheet, we see the following result:

```
01 C:\ptux\samp>java -jar ../prog/saxon.jar hello.xml hellohtm.xsl
02 <html>
03  <head>
04   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
05   <title>Greeting</title>
06  </head>
07  <body>
08   <p>Words of greeting:<br><b><i><u>Hello world.</u></i></b></p>
09  </body>
10 </html>
11 C:\ptux\samp>
```

Note how the end result contains a mixture of the stylesheet markup and the source instance content, without any use of the XSLT vocabulary. The processor has recognized the use of HTML by the type of the document element and has engaged SGML markup conventions.

The `<meta>` element on line 4 added by Saxon is ensuring the character set of the web page is properly recognized by conforming user agents.

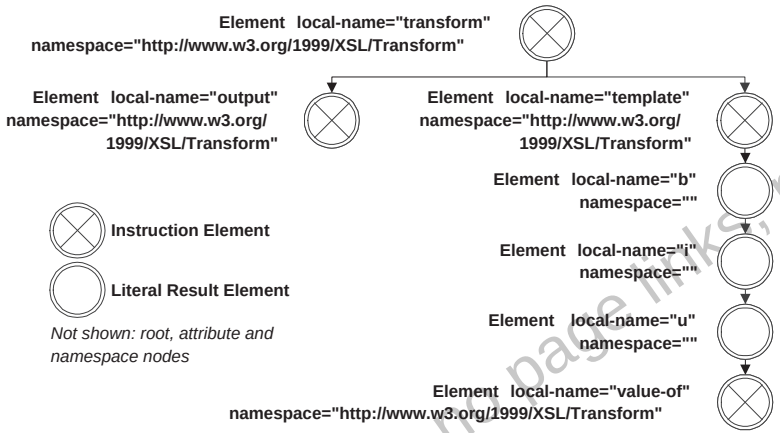
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Some simple examples (cont.)

Consider next the following XSLT file `hello.xsl` to produce XML output using the HTML vocabulary, where the output is serialized as XML:

```
01 <?xml version="1.0"?><!--hello.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03
04 <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05                version="1.0">
06
07 <xsl:output method="xml" omit-xml-declaration="yes"/>
08
09 <xsl:template match="/">
10     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
11 </xsl:template>
12
13 </xsl:transform>
```

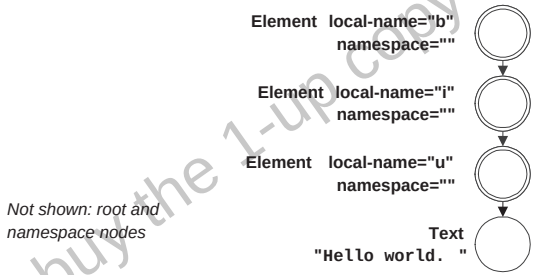
Remember that the syntax of the transform does not represent the syntax of the result, only the nodes of the result; the following is the node tree (not showing attribute and namespace nodes) of the stylesheet:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Some simple examples (cont.)

The node tree constructed using the stylesheet (page 50) on the source (page 48) is:

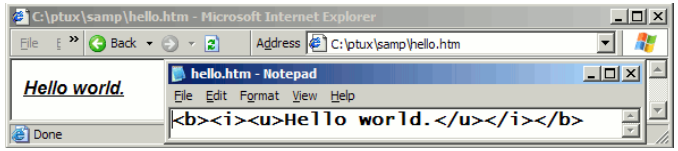


- note from the drawing conventions how the element nodes come from the operation node tree and the text node is calculated from the source node tree information

Using an MSDOS invocation to execute XSLT with the Saxon processor (with `-o` for the output file and `-a` to respect stylesheet association) we see the following tree serialization:

```
01 C:\ptux\samp>java -jar ../prog/saxon.jar -o hello.htm -a hello.xml
02
03 C:\ptux\samp>type hello.htm
04 <b><i><u>Hello world.</u></i></b>
05 C:\ptux\samp>
```

The result `hello.htm` file serialization of the constructed node tree can be viewed with a browser to see the results using the menu selection View/Source to examine the content:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

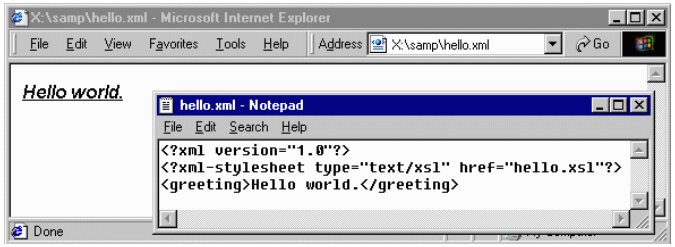
Some simple examples (cont.)

Two ways of working with the Microsoft XSLT processor:

Using the `msxml.bat` invocation batch file (documented in detail in free download preview of on-line tutorial material) at an MSDOS command line to execute the MSXML processor:

```
01 C:\ptux\samp>..\prog\msxml hello.xml hello.xsl hello-ms.htm
02 Invoking MSXML....
03
04 C:\ptux\samp>type hello-ms.htm
05 <b><i><u>Hello world.</u></i></b>
06 C:\ptux\samp>
```

Using IE to directly view the file will show the interpreted result on the browser canvas, in such a way that the menu function View/Source reveals the untouched XML:



Other browsers support on-the-fly XSLT transformation

- not all browsers have XML processors that support all of the syntax features of XML
  - e.g. lack of support for XML entities

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT stylesheet requirements

An XSLT stylesheet:

- must identify the namespace prefix with which to recognize the XSLT vocabulary
  - a typical namespace declaration attribute declares a particular URI for a given prefix:
    - `xmlns:`*`prefix`*`="http://www.w3.org/1999/XSL/Transform"`
    - as a common practice the prefix "`xsl`" is used to identify the XSLT vocabulary, though this is not mandatory
      - historically this is because XSLT was first published as one chapter of the XSL specification
      - all of the examples for XSL were written with "`xsl:`" and remained after XSLT was spun off as its own specification
    - the default namespace should not be used to identify the XSLT vocabulary
      - technically possible for elements of the vocabulary, but doing so prevents XSLT vocabulary attributes to be used wherever possible
      - not an issue for small stylesheets, but a maintenance headache if a large stylesheet needs to begin using XSLT attributes
  - extensions beyond the XSLT recommendation are outside the scope of the XSLT vocabulary so must use another URI for such constructs
- must also indicate the version of XSLT required by the stylesheet
  - this dictates the data model rules for building of the source tree based on XPath 1 or XPath 2
  - also engages the incompatible version-specific behavior of the processor for certain instructions
  - T2 using "`1.0`" for XSLT 2.0 either engages backwards-compatible behavior or signals an error and does not execute the transformation
  - in the start tag of an a element in the XSLT namespace
    - `version="`*`version-number`*`"`
    - attributes not in any namespace that are attached to an element in the XSLT namespace are regarded as being in the XSLT namespace
  - in the start tag of an element not in the XSLT namespace
    - *`prefix`*`:version="`*`version-number`*`"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT instructions and literal result elements

An XSLT instruction:

- is detected in the stylesheet tree only
  - not recognized if used in the source tree
  - instruction defined by the XSLT recommendation and specified using the prefix associated with the XSLT URI
- may be a control construct
  - the wrapper and top-level elements
  - procedural and process-control instructions
  - logical and physical stylesheet maintenance facilities
- may be a construction construct
  - synthesis of result tree nodes
  - copying of nodes from a source node tree
- may be a text value placeholder
  - any calculated value using `<xsl:value-of>` is replaced in situ
  - `<xsl:value-of select="greeting"/>`
  - this example instruction calculates the concatenation of all text portions of all descendents of the first of the selected points in the source tree
  - the `select=` attribute is an expression specifying the point in the source tree or, more generally, the outcome of an arbitrary expression evaluation which in this case is a node set
  - the value "`greeting`" indicates the name of a direct element child node of the current source tree focus, which at the time of execution in this example is the root of the document (hence `<greeting>` must be the document element)
- may be a custom extension
  - a non-standardized instruction implemented by the XSLT processor
  - implements extensibility
    - standardized fallback features allow any conforming XSLT processor to still interpret a stylesheet that is using extensions
  - specified using a namespace prefix associated with a URI known to the processor

A literal result element:

- any element not recognized to be an instruction
  - used in stylesheet file
  - any vocabulary that isn't a declared instruction vocabulary
- represents a result-tree node
  - element and its associated attributes are to be added to the result

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT templates and template rules

An XSLT template (a.k.a. "sequence constructor" in XSLT 2.0):

- specifies a fragment for constructing the result tree as a tree of nodes
  - see the nodes in Some simple examples (page 50)
- expressed in syntax as a well-formed package of markup
  - may or may not include XSLT instructions

```
01 <b><i><u><xsl:value-of select="greeting"/></u></i></b>
```

- a representation of the desired nodes to add to the result tree
- the XSLT processor recognizes any constructs therein from the XSLT vocabulary as XSLT instructions and acts on them
  - regards all other constructs not from the XSLT vocabulary as literal result elements that comprise a representation of a tree fragment to add to the result tree

An XSLT template rule:

- a result tree construction rule associated with source tree nodes
  - specifies the template to add to the result tree when processing a source tree node
  - a "matching pattern" describes the nodes of the source tree
  - see Extensible Stylesheet Language Transformations (XSLT) (page 23)

```
01 <xsl:template match="/">
02     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
03 </xsl:template>
```

- prepares the XSLT processor for building a portion of the result tree whenever the stylesheet asks the processor to visit the given source tree node
- uses the `match=` attribute as a "pattern" describing the characteristics of the source tree node associated with the given template
  - the pattern value "`/`" indicates the root of the source document (distinct from and the hierarchical parent of the document element of the source document, therefore, the very top of the hierarchy)
- a traditional stylesheet must declare all the stylesheet writer's template rules to be used by the XSLT processor
- a simplified stylesheet defines in its entirety the one and only template rule for the stylesheet, that being for the root node

XSLT processor first visits the source tree root node:

- the root node template rule begins the construction of the result tree
- *all* subsequent construction is controlled by the stylesheet

2 Can begin processing at a specified named template or mode at invocation

- source tree is optional when starting at an arbitrary rule

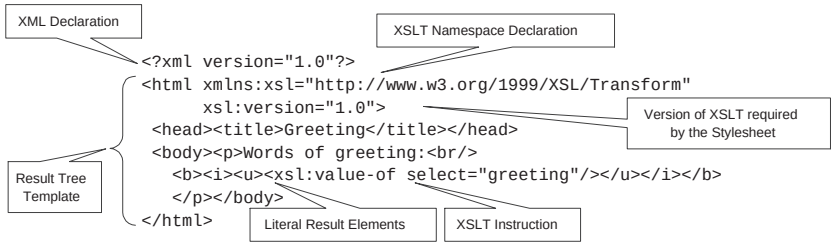
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT stylesheet components

Chapter 2 - Getting started with XSLT and XPath
Section 2 - Syntax basics

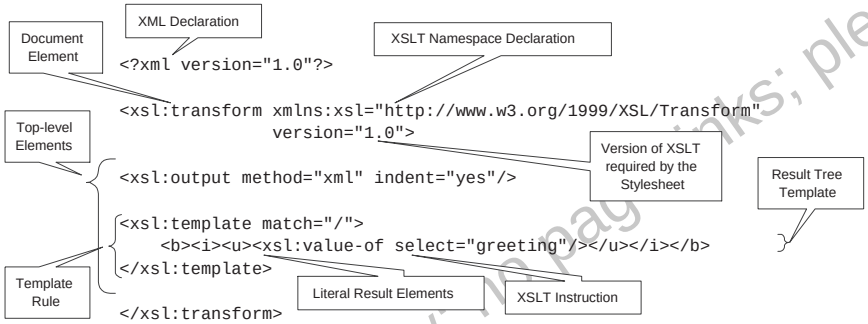
A "simplified" XSLT stylesheet:

- can be declared inside an arbitrary XML document (e.g. an XHTML document) by using namespace declarations for XSLT constructs found within
- the entire stylesheet file is a template for the entire result tree
  - regarded as the template rule for the root node
- identifiable components of this implicitly declared XSLT script:



A more traditional stylesheet:

- can be written as an entire XML document (or embedded fragment in an XML document) by using a stylesheet document element as the explicit container
- traditional stylesheets can be utilized by other explicitly declared stylesheets
- identifiable components of this traditional XSLT script:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Pull and push constructs

Chapter 2 - Getting started with XSLT and XPath
Section 3 - Approaches to transform design

Consider for illustration an XML file containing sale and purchase information maintained chronologically, thus in an arbitrary order:

```
<chrono-info>
 <purchase>...</purchase>
 <sale>...</sale>
 <sale>...</sale>
 <purchase>...</purchase>
 <sale>...</sale>
 <purchase>...</purchase>
</chrono-info>
```

How one approaches accessing the information to create the result tree varies

- one can pull the information out of the source node tree
- one can push the information from the source node tree at the stylesheet

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Pull and push constructs (cont.)

Pulling the input data and repositioning in the tree

- the hierarchy of the source file is known by the transform writer
- at the point of building "the next" part of the result tree

The transform "pulls" information as and when needed:

- from known locations in the source node tree
- for extraction or calculation using the lexical value
  - `<xsl:value-of select="string(`*XPath-expression*`)"/>`
- for extraction or calculation using the schema-qualified value
  - `<xsl:value-of select="`*XPath-expression*`"/>`
- for wholesale copying of source tree nodes
  - `<xsl:copy-of select="`*XPath-expression*`"/>`
- for repositioning over a sequence of arbitrary locations or values of any data type
  - `<xsl:for-each select="`*XPath-node-set-expression*`">`
    ...template...
    `</xsl:for-each>`
  - `<xsl:for-each select="`*XPath-sequence-expression*`">`
    ...template...
    `</xsl:for-each>`

Transform-determined result order implements direct document construction

- the result tree is built by the transform obtaining each result component from the source, in result order, and framing each component as required with literal markup from the transform
- if the result can be described as a single template using only the "pull" instructions, the transform can be simply declared as a single result template

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Pull and push constructs (cont.)

Pull example

- to process all of the sales records together, followed by all of the purchase records together, they are pulled from the source tree one set before the others:

```
- <xsl:template match="chrono-info">
    <sale-purchase-summary>
      <xsl:for-each select="sale">
        ...template for the sale...
      </xsl:for-each>
      <xsl:for-each select="purchase">
        ...template for the purchase...
      </xsl:for-each>
    </sale-purchase-summary>
  </xsl:template>
```

- each address "`sale`" and "`purchase`" will respectively find all `<sale>` and `<purchase>` child elements of `<chrono-info>`
- the order of the two instructions will construct the result tree by adding one template for each of the sale elements until all sale elements have been addressed, followed then by adding one template for each of the purchase elements until all purchase elements have been addressed:

```
- <sale-purchase-summary>
    ...result construction for sale...
    ...result construction for sale...
    ...result construction for sale...
    ...result construction for purchase...
    ...result construction for purchase...
    ...result construction for purchase...
  </sale-purchase-summary>
```

Recall the input elements shown on page 57

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Pull and push constructs (cont.)

Pushing the input data and repositioning in the tree

- arbitrary or unexpected source structure
  - the structure of the source file is not in either an expected or explicit order
  - source file order must be accommodated by transformation
- to process all of the records in the order they appear in the document, they are pushed through the transform logic while specifying the union of all such nodes

XSLT stylesheets:

- the stylesheet "pushes" nodes of information at the template rules:
  - visits known (using names) or unknown (using a wild card) source tree nodes
  - `<xsl:apply-templates select="`*XPath-node-expression*`">`
- template rules respond to node visitations by constructing the result tree:
  - `<xsl:template match="`*XPath-pattern*`">`

Source-determined result order implements indirect tree construction

- the `<xsl:apply-templates>` instruction is the event generators
  - selects the source information the processor is to visit
- the `<xsl:template>` template rule is the event handler
  - the type of event described as a qualification of the source information in the `match=` attribute

Note it is not necessary to exclusively use one approach or the other

- transforms alternately push some of the input data through the processor (data driven) and pull the same or other input data where required (transform driven)
- the pulling of data that is relative to the data being pushed can be done in the template catching the data being pushed

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Pull and push constructs (cont.)

Push example

- using XSLT:
  -
```
<xsl:template match="chrono-info">
  <sale-purchase-summary>
    <xsl:apply-templates select="sale | purchase"/>
  </sale-purchase-summary>
</xsl:template>
```
- where each construct being pushed must somehow be handled by the stylesheet:
  -
```
<xsl:template match="sale">...template...</xsl:template>
<xsl:template match="purchase">...template...</xsl:template>
```
- each address "`sale`" and "`purchase`" will respectively find all `<sale>` and all `<purchase>` elements in the source tree, but the union operator "`|`" will return the set of all those nodes in document order which may very well be interleaved
- the order of the two result expressions is irrelevant because each result expression will be triggered only by the kind of node being matched
- this will construct the result tree by adding one template for each of the sale elements and purchase elements in the document order encountered in the source tree:
  -
```
<sale-purchase-summary>
  ...result construction for purchase...
  ...result construction for sale...
  ...result construction for sale...
  ...result construction for purchase...
  ...result construction for sale...
  ...result construction for purchase...
</sale-purchase-summary>
```

Recall the input elements shown on page 57

Contrast the result with that of the pull approach on page 59

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Processing XML with many transforms

Consider the data file `prod.xml` containing some sales data information:

```
<?xml version="1.0"?><!--prod.xml-->
<!DOCTYPE sales [
<!ELEMENT sales ( products, record )> <!--sales information-->
<!ELEMENT products ( product+ )>         <!--product record-->
<!ELEMENT product ( #PCDATA )>      <!--product information-->
<!ATTLIST product id ID #REQUIRED>
<!ELEMENT record ( cust+ )>                <!--sales record-->
<!ELEMENT cust ( prodsale+ )>     <!--customer sales record-->
<!ATTLIST cust num CDATA #REQUIRED>     <!--customer number-->
<!ELEMENT prodsale ( #PCDATA )>     <!--product sale record-->
<!ATTLIST prodsale idref IDREF #REQUIRED>
]>
<sales>
  <products><product id="p1">Packing Boxes</product>
            <product id="p2">Packing Tape</product></products>
  <record><cust num="C1001">
            <prodsale idref="p1">100</prodsale>
            <prodsale idref="p2">200</prodsale></cust>
          <cust num="C1002">
            <prodsale idref="p2">50</prodsale></cust>
          <cust num="C1003">
            <prodsale idref="p1">75</prodsale>
            <prodsale idref="p2">15</prodsale></cust></record>
</sales>
```

Of note:

- each product is identified with `id=` declared to be of type `ID`
  - permits the element to be addressed with a unique identifier
- there is no total information, only information about each product sale
- the product names are not duplicated
  - the product information is referenced using `idref=` from the product sale

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing XML with many transforms (cont.)

The equivalent set of document constraints on the logical hierarchy expressed using W3C Schema could be in `prod.xsd`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="sales">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="products">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="product" maxOccurs="unbounded">
              <xsd:complexType mixed="true">
                <xsd:attribute name="id" type="xsd:ID"/>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="record">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="cust" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="prodsale" maxOccurs="unbounded">
                    <xsd:complexType>
                      <xsd:simpleContent>
                        <xsd:extension base="xsd:integer">
                         <xsd:attribute name="idref" type="xsd:IDREF"/>
                        </xsd:extension>
                      </xsd:simpleContent>
                    </xsd:complexType>
                  </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="num" type="xsd:string"/>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

- note the declaration of `prodsale` is an integer value
- note the ID/IDREF relationships expressed between `id=` and `idref=`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing XML with many transforms (cont.)

The hint that a particular W3C Schema applies to a document is given via reserved attributes

- a processor is not obliged to use the hints

The following document has a self-referential consistency error that will not be detected unless schema validation is turned on:

- note how customer C1003 has a product sale pointing to a non-existent product

In addition, the ID/IDREF relationships are not recognized unless schema validation is turned on:

- any built-in facilities for supporting ID-typed attributes are not engaged

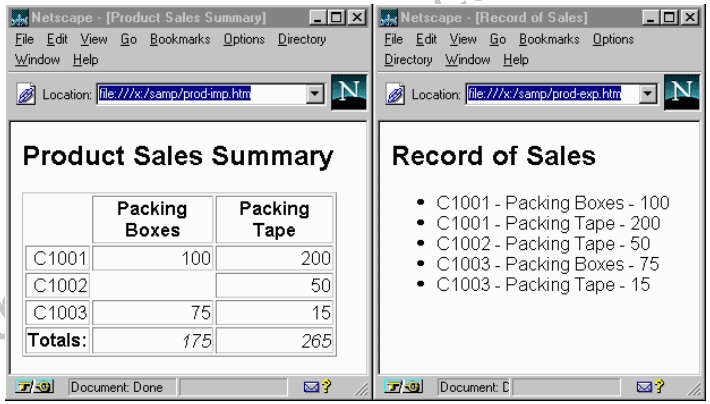
```
01 <?xml version="1.0"?><!--prod-bad.xml-->
02 <sales xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03        xsi:noNamespaceSchemaLocation="prod.xsd">
04   <products><product id="p1">Packing Boxes</product>
05             <product id="p2">Packing Tape</product></products>
06   <record><cust num="C1001">
07             <prodsale idref="p1">100</prodsale>
08             <prodsale idref="p2">200</prodsale></cust>
09           <cust num="C1002">
10             <prodsale idref="p2">50</prodsale></cust>
11           <cust num="C1003">
12             <prodsale idref="p1">75</prodsale>
13             <prodsale idref="p3">15</prodsale></cust></record>
14 </sales>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing XML with many transforms (cont.)

Recall the sample data on page 62

- very dissimilar reports could be generated for the one data file by using different transforms:



Of note:

- items are rearranged from one authored order to two different presentation orders
- transformation includes calculation of sum of marked up values

Copyright © Crane Softwrights Ltd.

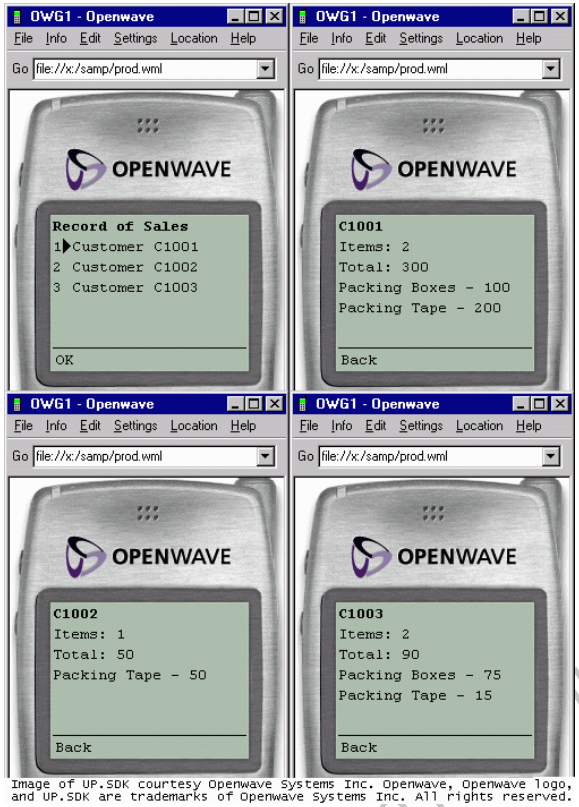
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Processing XML with many transforms (cont.)

Recall the sample data on page 62

- any result vocabulary can be used; for example, WML rendered on a mobile device:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Processing XML with many transforms (cont.)

Chapter 2 - Getting started with XSLT and XPath
Section 4 - More transform examples

The simplified stylesheet `prod-pull.xsl` for the table (page 65) for the XML (page 62):

```
01 <?xml version="1.0"?><!--prod-pull.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03 <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04       xsl:version="1.0">
05  <head><title>Product Sales Summary</title></head>
06  <body><h2>Product Sales Summary</h2>
07   <table summary="Product Sales Summary" border="1">
08                                          <!--list products-->
09    <tr align="center"><th/>
10     <xsl:for-each select="//product">
11      <th><b><xsl:value-of select="."/></b></th>
12     </xsl:for-each></tr>
13                                         <!--list customers-->
14    <xsl:for-each select="/sales/record/cust">
15     <xsl:variable name="customer" select="."/>
16     <tr align="right">
17      <td><xsl:value-of select="@num"/></td>
18      <xsl:for-each select="//product">  <!--each product-->
19       <td><xsl:value-of select="$customer/prodsale
20                                  [@idref=current()/@id]"/>
21       </td></xsl:for-each>
22     </tr></xsl:for-each>
23                                                <!--summarize-->
24    <tr align="right"><td><b>Totals:</b></td>
25     <xsl:for-each select="//product">
26      <xsl:variable name="pid" select="@id"/>
27      <td><i><xsl:value-of
28                 select="sum(//prodsale[@idref=$pid])"/></i>
29      </td></xsl:for-each></tr>
30   </table>
31 </body></html>
```

Information added from the stylesheet and pulled from the source document:

- the header and body title are hardwired content from stylesheet
- the table's header row comes from each product name in source
- the customer information is visited to produce rows (note use of variable)
  - sale information produces columns
- total information is generated by stylesheet using `sum()` built-in function (no custom node-traversal programming needed)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Processing XML with many transforms (cont.)

The traditional stylesheet `prod-push.xsl` for the list (page 65) for the XML (page 62):

```
<?xml version="1.0"?><!--prod-push.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="record">   <!--processing for each record-->
  <ul><xsl:apply-templates/></ul></xsl:template>

<xsl:template match="prodsale">   <!--processing for each sale-->
  <li><xsl:value-of select="../@num"/>   <!--use parent's attr-->
      <xsl:text> - </xsl:text>
      <xsl:value-of select="id(@idref)"/>      <!--go indirect-->
      <xsl:text> - </xsl:text>
      <xsl:value-of select="."/></li></xsl:template>

<xsl:template match="/">                          <!--root rule-->
  <html><head><title>Record of Sales</title></head>
    <body><h2>Record of Sales</h2>
      <xsl:apply-templates select="/sales/record"/>
    </body></html></xsl:template>

</xsl:stylesheet>
```

Source document is pushed through the stylesheet:

- the order of the template rules is irrelevant
  - only one node is being pushed at the stylesheet, so only one template responds
- the header and body title are hardwired content from stylesheet
- the root rule pushes all sales records through the stylesheet
- each record produces the `<ul>` unordered list wrapper for list items and pushes child elements through the stylesheet
- each child element pushed through produces a list item that pulls information from the parent and from an arbitrary place of the source

An importing stylesheet can exploit the template rule fragmentation

- another stylesheet can import this stylesheet and specialize the behavior of any top-level construct
- an overriding definition of any template rule will take precedence over that rule in the above transformation

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing XML with many transforms (cont.)

The traditional stylesheet `prod-wml.xsl` for the WML (page 66) and XML (page 62):

```
<?xml version="1.0"?><!--prod-wml.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output doctype-system="http://CRANE/wml13.dtd"/>

<xsl:template match="/">              <!--root rule-->
  <wml><card title="Record of Sales"> <!--index card-->
        <p><em>Record of Sales</em></p>
        <p><select name="cards">
             <xsl:apply-templates mode="head"
                  select="/sales/record/cust"/>
           </select></p></card>
   <xsl:apply-templates select="/sales/record/cust"/>
  </wml></xsl:template>

<xsl:template match="cust" mode="head"><!--index entry-->
  <option onpick="#{@num}">
    <xsl:text/>Customer <xsl:value-of select="@num"/>
  </option></xsl:template>

<xsl:template match="cust"><!--customer's card in deck-->
 <card id="{@num}" title="Customer {@num}">
  <p><em><xsl:value-of select="@num"/></em></p>
  <p>Items: <xsl:value-of select="count(prodsale)"/></p>
  <p>Total: <xsl:value-of select="sum(prodsale)"/></p>
  <xsl:apply-templates/></card></xsl:template>

<xsl:template match="prodsale"><!--proc for each sale-->
  <p><xsl:value-of select="id(@idref)"/> <!--indirect-->
     <xsl:text> - </xsl:text>
     <xsl:value-of select="."/></p></xsl:template>

</xsl:stylesheet>
```

Source document is pushed through the stylesheet:

- the same source is visited twice using different template rules for processing to produce different results

An importing stylesheet can exploit the template rule fragmentation

- another stylesheet can import this stylesheet and specialize the behavior of any top-level construct
- an overriding definition of any template rule will take precedence over that rule in the above transformation

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 3 - XPath data model

- Introduction - The need for abstractions
- Section 1 - XPath data model components
- Section 2 - XPath expressions

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The need for abstractions

Dealing with information, not markup

- all input and output information manipulated in an abstract fashion
- separate node structures of information:
  - source trees (example on page 48)
    - the main source tree is optional in XSLT 2
    - 1 the main source tree is required in XSLT 1
    - multiple additional source documents may be read as separate node structures
  - operation tree (stylesheet example on page 50)
  - result tree (example on page 51)
    - 2 multiple result trees can be created using XSLT 2
- knowledge of input markup and control of output markup out of the hands of the transform writer
  - the writer deals with nodes of information, not characters of markup

Traversing a source document or transform document predictably

- XML syntax processed into an abstract data model tree of nodes
  - documents are described according to a data model for the XML markup
    - interpreted in terms of the XML Information Set
      - `http://www.w3.org/TR/xml-infoset/`
    - 2 maintained in terms of a formal XPath data model
      - `http://www.w3.org/TR/xpath-datamodel/`
  - all nodes created are typed, and have a value that can be used as a string of text
  - some nodes have an associated name, while other nodes are unnamed
  - 2 some nodes have types based on W3C Schema post-schema validation infoset
    - `http://www.w3.org/TR/xmlschema-1/#d0e504`
  - the processor performs all operations using the node tree, not the document directly
    - the actual markup used in input instances is not preserved
    - there are no constraints or requirements of the XML source in that any well-formed markup chosen by the author of an XML document is represented abstractly in the tree
- XPath node tree navigation
  - the transform can navigate around the source node tree in many directions
  - thirteen axes of direction that can be traversed relative to the context (current) node
  - the transform is responsible for specifying which source nodes get processed when and how

The XPath data model and the DOM data model are different

- the Document Object Model (DOM) is a data model of the information including the syntax in an XML document
- XPath is a data model of the information not including the syntax in an XML document

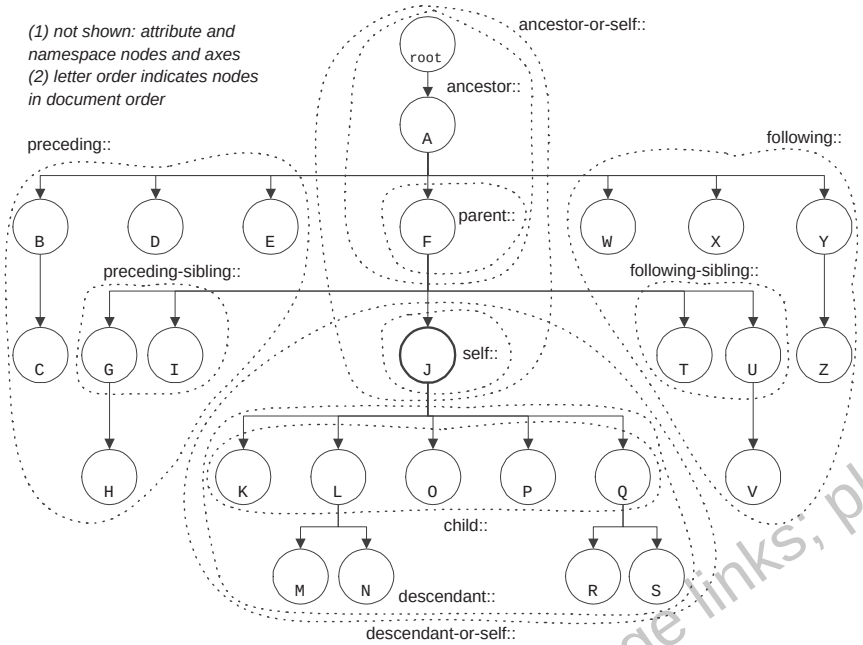
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The need for abstractions (cont.)

Consider an XML document comprised of 26 empty and non-empty elements named "A" through "Z" (without any text or new-line characters of any kind):

```
01 <A><B><C/></B><D/><E/><F><G><H/></G><I/><J><K/><L><M/><N/></L><O/>
02 <P/><Q><R/><S/></Q></J><T/><U><V/></U></F><W/><X/><Y><Z/></Y></A>
```

The following depicts this document's complete node tree (not showing namespaces):



- the root of the tree is at the top
- the leaves of the tree are towards the bottom
- the context node in this example is in the center with the bold circle
- the dotted lines completely surround the nodes of the tree that are members of each axis relative to the context node

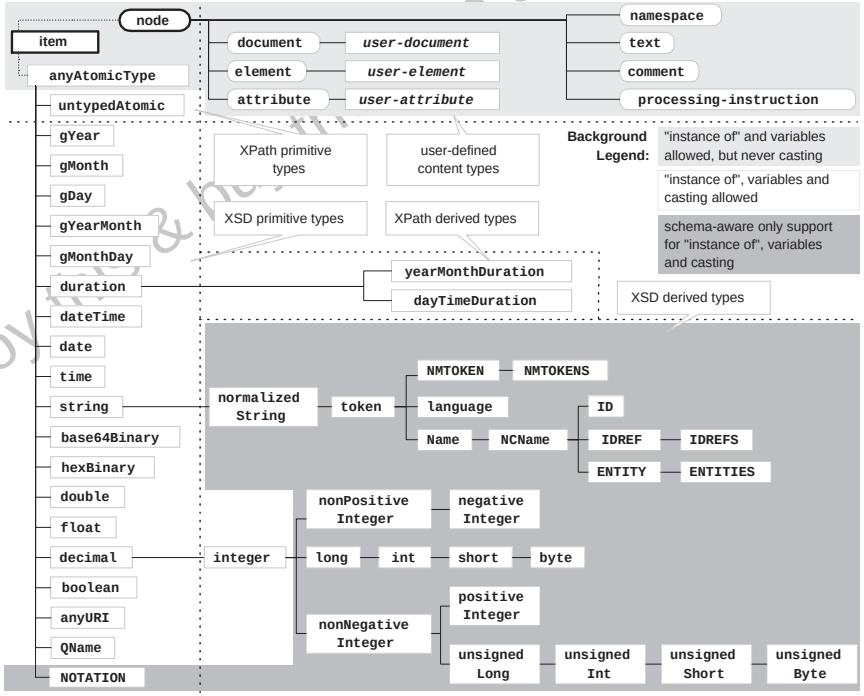
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data types

P1 XPath 1.0 treats node values as strings and has a limited number of data types

- boolean, number, string, node set and result tree fragment

P2 XPath 2.0 introduces data types based on the W3C Schema data type hierarchy:



When referenced in transforms, types must be namespace qualified:

- e.g. `xmlns:xs="http://www.w3.org/2001/XMLSchema"`
  - any prefix can be used

P2 Types allowed for casting are value constructors when used with function syntax:

```
- <xsl:variable name="meetingStart" as="xs:dateTime"
                select="xs:dateTime('2005-12-04T11:00:00Z')"/>
```

P2 Some types are only available in schema-aware versions of the processor

- e.g. `NOTATION` and most XSD derived types

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sequence types

A sequence type is a declaration combination of data type and cardinality

- `item()` - union of any node type or atomic value
- `node()` - any node (tree construction - see page 119)
  - seven types of tree nodes described in this chapter
  - e.g. `element()`, `attribute()`, `text()`, etc.
  - three types of named tree nodes described in this chapter
  - e.g. `element(`*`name`*`)`, `attribute(`*`name`*`)`, `processing-instruction(`*`name`*`)`
  - two types of typed tree nodes described in this chapter
  - e.g. `element(`*`name`*`,`*`type`*`)`, `attribute(`*`name`*`,`*`type`*`)`
  - two types of declared tree nodes described in this chapter
  - e.g. `schema-element(`*`name`*`)`, `schema-attribute(`*`name`*`)`
  - qualified document nodes described in this chapter
  - e.g. `document-node(`*`element-name-type-declaration-test`*`)`
  - user-defined globally-declared types
  - e.g. `mySchema:zip-code`
- `xs:anyAtomicType` - any atomic value (lexical construction - see page 73)
  - W3C Schema data types
  - e.g. `xs:string`, `xs:integer`, `xs:duration`, `xs:gMonth`, etc.
  - XPath 2 data types
  - e.g. `xs:dayTimeDuration`, `xs:yearMonthDuration`
  - e.g. `xs:untypedAtomic` - an atomic value without a type

Non-zero cardinality specified using Kleene operators "`+`", "`?`" and "`*`", for example:

- `empty-sequence()` - zero to zero (i.e. no items of any kind)
- `xs:string` - exactly one string (i.e. mandatory)
- `xs:string?` - zero or one strings (i.e. optional)
- `xs:string+` - one or more strings (i.e. mandatory and repeatable)
- `xs:string*` - zero or more strings (i.e. optional and repeatable)
- `element( email )+` - one or more `<email>` elements
- `element( email )?` - zero or one `<email>` elements
- `xs:item()*` - zero or more items of any type

The following data types are not allowed as sequence types

- `xs:anyType` (un-validated element node)
- `xs:untyped` (un-validated attribute node content)
- `xs:anySimpleType` (lists and unions only)
  - includes `xs:IDREFS`, `xs:NMTOKENS` and `xs:ENTITIES`
  - includes user-defined list and union types

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result trees

Building a result predictably

- created as an abstract tree of nodes
  - the result node tree is constructed using nodes from the operation and source trees, and nodes synthesized by operation tree expressions
  - result is described according to the same data model for XML markup as is used for input
  - the processing of a template builds a portion of the result as sub-tree of nodes reflecting the output information
  - the interpretation of a result template is reliable and reproducible
    - the processor acts on instructions the same way every time
    - some aspects (e.g. order of attribute) is implementation-dependent
- one-pass construction of the result tree
  - no "going back and changing" anything in the result tree
  - created in a single pass in result parse-order
- serialization instantiates markup syntax
  - the emission of the result node tree (if so desired)
  - in XML markup according to the standard
    - the transform writer does *not* control which markup constructs are used for representing XML information
    - the processor can make arbitrary decisions as long as the end result is well formed
  - using alternative markup or syntax conventions if made available by the processor (e.g.: interpretation of a colloquial vocabulary into a binary format)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath data model

Chapter 3 - XPath data model

The XPath productions covered in this chapter are:

- `(: :)`
  - commenting XPath expressions
- `if ( ) then else`
  - XPath choice statement
- `for ... return ...`
  - XPath tuple statement
- `empty-sequence()`
  - the empty sequence sequence type
- `/`
  - location path address steps
- `$`
  - variable references
- `( )`
  - parenthesized expressions

The XSLT instructions covered in this chapter are:

- `<xsl:strip-space>`
  - indicate those source tree nodes in which white-space-only text nodes are not to be preserved.
- `<xsl:preserve-space>`
  - indicate those source tree nodes in which white-space-only text nodes are to be preserved

The XPath functions covered in this chapter are:

- `last()`
  - the number of nodes in the context/current node list
- `position()`
  - the ordinal number of the current node in the context/current node list
- `.`
  - context item

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The file abstractions

Chapter 3 - XPath data model
Section 1 - XPath data model components

XPath describes 7 kinds of nodes present in node trees (illustrated on page 72)

- some node kinds are not visible in certain types of node trees
- some node kinds are only tree branches, or only tree leaves, or can be both
- some node kinds are considered children
  - those that are children are ordered in the node tree in document order
  - those that are not children are ordered in any order chosen by the processor
- some node kinds are named
- all node kinds have a value of some type
  - either directly associated with the node or calculated from descendent node values
  - simple lexical strings directly reflecting the characters in the XML syntax
    - XML normalization is done on end-of-line sequences and DTD attribute types
    - P2 the data type of the string value is `xs:untypedAtomic`
  - P2 both the lexical XML and the XSD schema-qualified values are available
    - in the presence of a W3C Schema declaration for the node and validation is engaged
    - consider an example where `<mynum>` is declared in the schema to be a number
    - e.g. `<mynum>0001</mynum>` has a string value of "`0001`"
      - the string representation of the value is the "schema normalized value"
    - e.g. `<mynum>0001</mynum>` has a data value of `1`
      - the data representation of the value is the "schema typed value"
- all node kinds have a basic addressing syntax in an XPath expression
  - detailed later in the module with nuances and alternative syntaxes
- P2 formal details for XPath 2: `http://www.w3.org/TR/xpath-datamodel/`

Every node has an absolute "base URI" (illustrated on page 9)

- the URI of the physical entity in which the entity was parsed
  - either the document's URI or an external entity's URI
- used when dealing with relative URI values in attached and descendent nodes
- P2 declarations of `xml:base=` are respected
  - `http://www.w3.org/TR/xmlbase/`

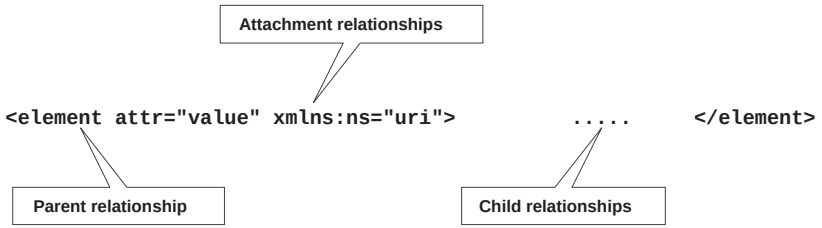
Every node has a unique generated identifier persistent only within the transformation

- comprised of only alphanumeric characters and no punctuation
- composed by any method the processor wishes to use and is opaque to user dissection
- unique across all nodes of all source trees and fixed only for the duration of the transform
  - may be different the next time the same tree is used in the same transform

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Parent/child and attachment relationships

The relationships between nodes:



- an attached node is a node created from markup found in the start or empty tag of an element
  - specified and defaulted attributes
  - namespaces
- a child node of an element node is a node created from markup found between the start and end tag of an element
  - text
  - comments
  - processing instructions
  - elements

Root node is at the top of the node tree

- called the "document node" in XPath 2.0
- children include the document element
- other children are the comments or processing instructions in the prologue and epilogue of the document (respectively the markup before the start of the document element and after the end of the document element)
- the only text outside of markup declarations outside of the document element is white-space and is not represented in the node tree

A node is the parent of all nodes that are attached to it or are children of it

- the root is the only node in the tree without a parent
  - exception that the node of a variable of a single tree node also does not have a parent

Document order is an important concept (illustrated on page 72)

- hierarchical XML components are ordered in a "depth-first, breadth next" order
- also called "parse order" (easily remembered as "the order of the start tags")
- some constructs are inherently unordered according to XML
- at times nodes can be counted in "reverse document order"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Comment node and processing instruction node

XML annotations are extrinsic information, not constrained by a document model:

Comment node

- a single unnamed node is created from a single comment
  - `<!--`*`comment-value-string`*`-->`
- this type of node is a child of its parent node
- this type of node is always a leaf of the tree and never has any child nodes of any type
- the node value is the content of the comment not including the opening and closing markup delimiters
- it is an error if a created comment node is supplied with the "`--`" text string or ends with a "`-`" character
  - an XSLT processor may choose to not report the error and separate the two characters with a space or follow the final character with a space
- addressed using: `comment()`

Processing instruction node

- a single named node is created from a single processing instruction
  - `<?`*`piTarget`*`?>`
  - `<?`*`piTarget p-i-value-string`*`?>`
  - the name of the node is the XML Processing Instruction Target
    - no access to any NOTATION that may be formally declaring the PI target
- this type of node is a child of its parent node
- this type of node is always a leaf of the tree and never has any child nodes of any type
- the node value is the string content of the processing instruction following all white space following the name, but not including the opening and closing markup delimiters
  - there are no attribute nodes for pseudo-attributes in PI values (see Stylesheet association (page 34) for an example)
  - the value is always a simple string
  - all white space after the first non-white-space character is preserved (no normalization)
- it is an error if a created processing instruction node is supplied with the "`?>`" text string
  - an XSLT processor may choose to not report the error and separate the two characters with a space
- addressed using: `processing-instruction(`*`optional-target`*`)`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Element node

- a single node is created for each element
  - <*elementName* *attributes*/>
    - an empty element using the empty element tag
  - <*elementName* *attributes*></*elementName*>
    - an empty element using a start tag and an end tag
  - <*elementName* *attributes*>*content*</*elementName*>
    - a non-empty element using a start tag and an end tag
  - the same order as the elements' start tags in the instance
  - depth-first breadth-next tree order (a.k.a. document order or parse order)
  - supports processing predictability
    - many XPath facilities address document nodes in document order
    - some facilities address document nodes in reverse document order
- this type of node is a child of its parent node
- may have *child* content nodes
  - text, element, comment and processing instruction nodes
  - descendants of an element node are considered to be all the child nodes plus the descendants of child element nodes
  - does not directly include any character data as any element's character data is found only in a child text node
  - an empty element does not have any child nodes
- may have *attached* non-content nodes
  - *not* considered to be child nodes of an element node
  - the element node *is* considered to be the parent of an attached node
  - specified or defaulted attributes as attribute nodes
  - namespace declaration nodes
  - an empty element may have attached nodes
- may have a user-defined unique identifier useful for random access
  - by way of an attribute of type `ID` as detected by the XML processor
    - P2 by way of an element's content being of type `ID`
  - this is different than the vendor-generated unique identifier for every node
  - scope of uniqueness for user-defined identifiers is only the given source tree
  - only guaranteed to be unique if the document is validated
  - nodes distinguished by their unique identifier are returned from calling a function, not from using an axis
- the name of the node is the element type
- the value of the node is the concatenation, in document order, of all text nodes that are descendants of the element
- addressed using: `elementName` or the wildcard `*`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespace node

- the value of the URI declared to use in place of the namespace prefix
  - `xmlns="`*`default-namespace-URI`*`"` and `xmlns=""`
  - `xmlns:`*`namespace-prefix`*`="`*`namespace-URI`*`"`
  - `xmlns:`*`namespace-prefix`*`=""` (XML 1.1 only)
- an element has as many namespace nodes as it and its ancestors have declarations
  - can never be used in a matching pattern, only in a selection expression
  - present in the source and operation trees and the processor adds them to the result tree when copying element nodes
    - a common problem when copying nodes to the result tree is the copying of unwanted namespace nodes that originate from ancestral declarations
  - `xmlns:xml="http://www.w3.org/XML/1998/namespace` on every node
- consider an example where `e1` has two namespace nodes attached and `e2` has three

```
01 <e1 xmlns:p="urn:X-P">
02   <e2 xmlns="urn:X-Q" p:abc="abc" def="def" xml:id="x"/>
03 </e1>
```

  - namespace node for `xml:` on both elements
  - `e1` and `def=` are in no namespace
  - `e2` is in the "`urn:X-Q`" namespace; `p:abc=` is in the "`urn:X-P`" namespace
- namespace nodes are arbitrarily ordered
- this type of node is *attached* to element nodes and *not* considered a child
  - the element to which it is attached *is* considered its parent
  - P1 it is always attached to an element
  - P2 attached to an element when attached, but it may be standalone in a variable
- this type of node is always a leaf of the tree and never has any child nodes of any type
- the name of the node is the namespace prefix
- the value of the node is the namespace URI
- copying any source or operation element to the result copies all its namespace nodes
- operation tree literal result elements can have namespace nodes pruned
  - `[xsl:]exclude-result-prefixes="`*`space-separated-prefixes`*`"`
  - prunes the given namespace node from descendent literal result elements
  - the namespace prefix is needed only if this attribute is placed on a literal result element, not when placed on an XSLT instruction
- the prefix used in a transform to refer to a namespace URI need not be the same prefix used in a source document for the URI being referred to
  - each document and associated node tree has own set of prefixes
  - node naming based upon referenced URI values in each independent prefix space
  - common mistake to try to match a default non-null namespace source tree element with a default null namespace operation tree element or expression
- addressed using: `namespace::`*`prefix`* or the wildcard `namespace::*`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Attribute node

Chapter 3 - XPath data model
Section 1 - XPath data model components

- a single attribute node is attached to an element node for
  - each attribute specified in the start tag of that element
    - *attributeName*="*specifiedString*"
    - *attributeName*='*specifiedString*'
  - each attribute with a default value in the document model
    - <!ATTLIST *elemType attrName attrType* "*defaultString*">
    - <!ATTLIST *elemType attrName attrType* '*defaultString*'>
    - for a document model expressed in a DTD (Document Type Definition), a node is *not* created for an attribute declared with an #IMPLIED default value
    - <xsd:attribute name="*attrName*" default="*defaultString*">
    - <xsd:attribute name="*attrName*" default='*defaultString*'>
    - only schema-aware processing recognizes W3C schema defaults
- attribute node contents normalized according to XML rules
  - XML section 2.11 changes any end-of-line sequence into a linefeed character
  - XML section 3.3.3 changes any attribute linefeed character into a space
  - XML section 3.3.3 changes sequences of white space characters in non-CDATA attributes to a single space
    - a specified attribute without a DTD declaration is assumed to be CDATA
- attribute nodes are arbitrarily ordered
- this type of node is always *attached* to element nodes
  - it *is not* considered a child of the element node
  - the element to which it is attached *is* considered its parent
- this type of node is always a leaf of the tree and never has any child nodes of any type
- the name of the node is the name of the attribute
- the value of the node is the normalized value of the attribute (as described above)
- addressed using: @*attributeName* or attribute::*attributeName* or the wildcards @* or attribute::*

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Attribute node (cont.)

Chapter 3 - XPath data model
Section 1 - XPath data model components

- the XML Namespace attributes xmlns= and xmlns:*prefix*= *are* treated specially
  - creates a namespace node *only*, an attribute node is *not* created
- the XML attributes xml:space= and xml:lang= *are not* treated specially
  - it is the transform's responsibility to copy or create such nodes in the result tree
  - the presence of such attributes in XSLT instructions in the stylesheet does not implicitly reproduce the use of the attributes in the result
- no "global" attributes
  - in XML Recommendation 1.0 there is no such construct as a "shared attribute" or "global attribute", which would be associated with more than one instantiated element or defined element type
    - a defaulted or specified attribute of any given element type is instantiated for *every* such element in the instance
    - a defaulted or specified attribute that may happen to have the identical declaration in each of two element types is not treated specially and is instantiated once for every element of each type in the instance

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Attribute node (cont.)

- an attribute of *type* `ID` is significant
  - confers a unique identifier to the element
    - the element is assigned a unique identifier from the attribute value
    - requires the source file to have a document model whose declaration for the attribute is of type `ID`
      - no special meaning conferred for attributes named "`id`", only for attributes of type `ID`
      - note that it is an error if there are two elements in the source tree with both an attribute of type `ID` and the same value for that attribute
        - if the processor chooses to not report the error, the first element with the attribute of the given value is considered the element with the unique identifier
  - well-formed document examples:
    - without a complete DTD it is sufficient to add only an attribute list declaration for the attribute in question

```
<!DOCTYPE prodsummary [
<!ATTLIST product id ID #REQUIRED>
]>
```

and

```
<!DOCTYPE custsummary [
<!ATTLIST cust custNbr ID #REQUIRED>
]>
```

and

test.dtd:

```
<!ATTLIST cust custNbr ID #REQUIRED>
```

test.xml:

```
<!DOCTYPE custsummary SYSTEM "test.dtd">
```

  - W3C schema example declaration:

```
01 <xsd:element name="product" maxOccurs="unbounded">
02   <xsd:complexType mixed="true">
03     <xsd:attribute name="id" type="xsd:ID"/>
04   </xsd:complexType>
05 </xsd:element>
```

- recognizes `xml:id=` to implicitly have an `ID` type
  - `http://www.w3.org/TR/xml-id/`
- it is a common error to process an input file that does not declare `ID` typed attributes with a transform expecting to use unique identifiers
  - function calls returning the sought nodes return the empty node set

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Text node

- a single unnamed node is created for a string of adjacent character data
  - all text and character markup that is not any other kind of document markup
    - `<e1>`*`text1`*`<e2>other</e2>`*`text2`*`<!--c1-->`*`text3`*`</e1>`
      - `e1` has mixed content: a total of five children: three text nodes, element `e2` and the comment
      - note the text "`other`" is a child of `e2` not a child of `e1`
        - "`other`" is a descendent text node of `e1`
    - characters are regarded in their abstract UCS (Universal Character Set) or Unicode value, not by any entity that may have been used to specify the character in the instance
      - the text node below `<elem>abc<def</elem>` is "`abc<def`"
      - impediment to supporting markup preservation in transforms
        - the markup used to represent characters is not preserved
    - the processor can choose to serialize result tree text node characters using any escaping mechanisms appropriate to the active serialization method
  - even element content can have sibling text nodes
    - `<price>30<!-- illegible, might be 39 -->.00</price>`
    - the `<price>` element has three child nodes
  - all entity references are expanded and their boundaries are not preserved
    - all text is treated at the end result of parsing the instance
  - all text not in markup before or after the document element is ignored
    - by definition it must be white space and is, therefore, not considered part of the information
  - see important detailed notes later in this chapter regarding white space preservation in text nodes
- any CDATA section boundaries are *not* preserved
  - considered as only syntactic sugar that reduces the amount of text escaping done in the creation of XML textual information
  - all text found in a CDATA section is added to the text node and is indistinguishable from text not found in a CDATA section
- line endings from external entities are normalized to `
` per XML 1.0 section 2.11
- note that characters found inside comments and processing instructions are not considered character data, thus they are not captured in text nodes in the source tree
  - note, however, that text nodes can be and are typically used in the transform to specify the content of result tree comments and processing instructions
- this type of node is a child of its parent node
- this type of node is always a leaf of the tree and never has any child nodes of any type
- a text node's value is considered a sequence of Unicode characters
  - unlike a string, a copied text node is not separated by a space from its neighbor
- addressed using: `text()`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

White-space-only text nodes

Text nodes with any non-white-space characters are always preserved and are not treated specially

- all white space is preserved in every text node found to have at least one character other than `#x20`, `#x9`, `#xD` or `#xA` (as defined by XML; respectively these characters are space, horizontal tab, carriage return, and line feed)

White-space-only text nodes are often used for indentation

-
```
<elem1>
  <elem2/>
  <elem3>
    <elem4/>  <elem5/>
  </elem3>
  <elem6>  </elem6>
</elem1>
```
  - this example has white-space-only text nodes for indentation (`elem1` and `elem3`) and white-space-only text nodes for content (`elem6`)
- indentation is often important to humans but is usually never important to XML processors
- a transform written for data produced from an XML processor usually needn't be careful about white-space if the source processor is emitting tightly expressed content
- a transform written for input instances authored by people need to be sensitive to the vagaries of human data entry patterns
  - some people will specify an "empty element" using start and end tags on different lines not realizing there is a text node between the two tags

White-space-only text nodes in the operation tree ("boundary space"):

- white-space-only text strings within and between XPath expressions are not significant
- preservation is a local property of text in an XSLT stylesheet
  - each individual white-space only text node is either preserved or discarded
  - the use of the XML reserved attribute `xml:space=` in either the source or transform instances can indicate which white-space-only text is significant

White-space-only text nodes in the source tree:

- F2 the invocation of XPath 2.0 can specify behavior
  - which white-space-only text in the source file creates white-space-only text nodes in the source tree
- T1 T2 an XSLT stylesheet can specialize white-space-only text node preservation on a per element basis
- the use of the XML reserved attribute `xml:space=` in either the source or transform instances can indicate which white-space-only text is significant

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

White-space-only text nodes (cont.)

When white-space-only text between instructions, expressions or construction nodes are not included in the operation tree

- it is assumed to be indentation in the transform
- the writer of the transform must protect isolated strings of white-space that are needed in the operation tree, otherwise they will be ignored in the creation of the operation tree

Preserved in the operation tree by the XSLT processor

- there is no XSLT keyword to specify that white-space-only text nodes are preserved
-
```
<xsl:value-of select="given"/> <xsl:value-of select="surname"/>
```
  - intuitively one would think the end result would put a space between the two values
  - incorrect because the space is ignored and the two string values are abutted
  - using `&#x20;` doesn't help because it is translated to a space by the XML processor
-
```
<xsl:value-of select="given"/>
<xsl:text> </xsl:text>
<xsl:value-of select="surname"/>
```
  - all white space is preserved in every text node that is a child of an `<xsl:text>` instruction found in stylesheet instance
  - note that `<xsl:text>` elements cannot have any elements in their content (e.g. XSLT instructions or literal result elements), they can only have text
-
```
<xsl:value-of select="surname"/>, <xsl:value-of select="given"/>
```
  - not a problem because the text between expressions is not white-space-only

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

White-space-only text nodes (cont.)

P2 Controlled in the source by the invoker of the transformation
- strip all white-space-only text nodes
- strip ignorable white-space-only text nodes
  - those that are in element content, not those in mixed content
- strip no white-space-only text nodes

Controlled in the source by the writer of the transformation
- the transform writer should assume the most difficult case of there being white-space-only text children anywhere
- P1 all element types are assumed to have white-space-only child nodes preserved
  - the list of element types for which white-space-only child text nodes are stripped is initially empty
- P2 all element types should be assumed to have white-space-only child nodes
  - the transform writer should write defensively by assuming all element types have white-space-only child nodes present and preserved
  - no safer assumption can be made about all element types because of invocation
- XSLT provides some stylesheet assertions regarding source tree creation
  - `<xsl:strip-space elements="`*`qnames-or-*`*`"/>` changes assumptions
    - grows the space-separated list in `elements=` of element types being stripped of white-space-only child text nodes
  - `<xsl:preserve-space elements="`*`qnames-or-*`*`"/>` overrides changed assumptions
    - shrinks the space-separated list in `elements=` of element types being stripped of white-space-only child text nodes
    - the ability to grow the list after shrinking the list or shrink the list after growing the list is made available for stylesheets that are imported to other stylesheets
    - the precedence rules for these instructions are the same as those for `<xsl:template>` described for `<xsl:import>` in Chapter 6 Transform and data management (page 209)
- applicable for only given element's direct child text nodes, not other descendent text nodes

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

White-space-only text nodes (cont.)

Overridden by the author of an XML instance
- XML 1.0 reserved attributes can be used by the author of an instance
- at the data model layer
  - `xml:space="preserve"` specifies that white-space-only nodes be preserved
  - `xml:space="default"` requests no special handling of white-space-only nodes
  - no impact at data model layer on the white-space found in non-white-space-only text nodes
- at an application/interpretation layer
  - `xml:space="preserve"` can be used to indicate no handling of white-space found in non-white-space-only text nodes
  - e.g. for poetry or for mimicking HTML `<pre>` element for preformatted content
  - up to the transform writer to decide what to do with non-white-space-only text nodes in the influence of `xml:space="preserve"`
- applicable for all given element's descendent text nodes
  - to the point at which a descendent element includes another white space XML declaration
  - according to the XML 1.0 recommendation, any XML document can declare the intent that white space in document content be preserved by applications by using `xml:space="preserve"`, or can declare the application to do what it wishes with white space by using `xml:space="default"`:
- white-space-only text nodes under the influence of `xml:space="preserve"` are preserved regardless of transform-specified controls
  - overrides any other assumptions made by the processor
- can be useful in XSLT stylesheets to preserve the stylesheet indentation when constructing the result tree
  - e.g. when an XSLT stylesheet is creating an XSLT stylesheet

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Internet Explorer compatibility

Implementation compatibility warning re: Internet explorer

```
C:\ptux\samp>type task.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="task.xsl"?>
<task>
 Performed by: <name><given>Jane</given> <surname>Doe</surname></name>
</task>

C:\ptux\samp>..\prog\xsltjavasaxon task.xml task.xsl task.htm
Transforming task.xml with task.xsl to task.htm with XSLT 1.0...

C:\ptux\samp>type task.htm
<html>
   <body>
      Performed by: <i>Jane</i> <i>Doe</i>

   </body>
</html>
C:\ptux\samp>..\prog\msxml task.xml task.xsl task.htm
Invoking MSXML....

C:\ptux\samp>type task.htm
<html>
<body>
 Performed by: <i>Jane</i><i>Doe</i></body>
</html>

C:\ptux\samp>
```

- the XSLT specification uses the XPath data model and does not say anything formally about white-space text nodes
- *most* XSLT processors preserve all white-space-only text nodes
- the typical processing of white-space-only text nodes is exhibited by the Saxon processor producing a result with a space between the transformed given name and surname
- the default setting for the Microsoft XML processor is to ignore white-space-only text nodes in both mixed content and element content in the data model of the source document
  - loss of white-space between `</given>` and `<surname>`
- the Microsoft XSLT processor can be engaged through an API to preserve white-space-only text nodes, but Internet Explorer provides no options to do so

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document node

- P1 a.k.a. "root node"
- the only node in the tree without a parent
  - the root node only occurs at the root of the tree as the parent of the document element
  - P2 exception that the node of a variable of a single tree node also does not have a parent
- source tree child nodes are restricted
  - no text nodes
  - exactly one element node (the document element)
  - any number of nodes needed for comments and processing instructions present in the XML instance outside of (either before or after), and in order with the element node for the document element
- result tree child nodes are not restricted
  - can include any number of element, text, comment and processing instruction nodes
  - the type and count of nodes dictates the serialization of the result:
    - a well-formed document entity
      - with identical constraints as found in the source tree
    - a well-formed external parsed general entity
      - the constraint of a single element node as a child of the root node is relaxed
- the node is not named
- the node's value is the value of the document element
- addressed using: `/`

Recall the physical hierarchy of XML documents (slide 9)

- a transformation can be used in a strategy of harvesting information from XML documents into external parsed general entities that are brought into the definition of other XML documents

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of XPath data model nodes

The following summarizes how these node types compare in general:

Node type	Visible in node tree	Tree position	Child/ Attached	Named	Valued
Element	all node trees	leaf (empty) or branch (not empty)	child	element type name	string of all descendent text nodes concatenated
Attribute	all node trees	leaf	attached	attribute name	specified or defaulted string
Namespace	depends*	leaf	attached	prefix	URI string
Processing Instruction	not in operation node tree	leaf	child	PI Target	string after all first white-space
Comment	not in operation node tree	leaf	child	no	string
Text (white-space-only)	depends**	leaf	child	no	Unicode sequence
Text (non-white-space-only)	all node trees	leaf	child	no	Unicode sequence
Document (Root)	all node trees	branch	N/A	no	document element

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of XPath data model nodes (cont.)

*Namespace node dependencies:

- duplicated in all element nodes descendent from the declaration of the namespace
- pruned from the stylesheet tree during tree creation by using:
  - `xsl:exclude-result-prefixes="`*`names`*`"`
- cannot be pruned from the source tree during creation
- copied to the result tree when copying elements from the source tree or operation tree

**White-space-only text node dependencies:

- not made present in the operation tree for XSLT unless it is within the instruction:
  - `<xsl:text> </xsl:text>`
- at user request for the source tree
  - P2 at processor invocation
- can be pruned from the source tree during tree creation by using:
  - `<xsl:strip-space elements="`*`qnames-or-*`*`"/>`
- source tree pruning can be overridden to be preserved by using:
  - `<xsl:preserve-space elements="`*`qnames-or-*`*`"/>`
- can be preserved in any XML document during tree creation when a descendant of an element using the attribute:
  - `xml:space="preserve"`
- can always be in the result tree

Specifically not included in the XPath data model but found in the Document Object Model (DOM):

- CDATA section nodes
  - considered to be only syntactic sugar and not part of the information structure
- document fragment and entity reference nodes
  - syntactic fragmentation is consumed by the XML processor within the transformation processor
- document type, entity and notation nodes
  - no such information in XPath data model
  - very limited access to unparsed entity URI strings through XSLT

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Depiction of a complete node tree

Consider the well-formed XML instance `partlist.xml`:

```
<?xml version="1.0"?>
<!--start-->
<part-list bin="78"><part nbr="A123">bolt</part>
<part nbr="B456">washer</part><warn level="1" type="max"/>
<!--end of parts--><?cursor   blink  under ?>
</part-list>
```

A summary of each of the types of nodes in this instance and their corresponding example values is as follows (other columns related to expressions are explained later in this chapter):

Node type	Node without specificity	Node in isolation	Isolation example	Example value (inside quotes)
Element	`*`	*`elementName`*	`part`	`"bolt"`
Attribute	`@*` or `attribute::*`	`@`*`attributeName`* or `attribute::`*`attributeName`*	`@nbr`	`"A123"`
Namespace	`namespace::*`	`namespace::`*`prefix`*	`namespace::xml`	`"http://www.w3.org/XML/1998/namespace"`
Processing Instruction	`processing-instruction()`	`processing-instruction(`*`piTargetString`*`)`	`processing-instruction('cursor')`	`"blink  under "`
Comment	`comment()`	N/A	N/A	`"end of parts"`
Text	`text()`	N/A	N/A	`"bolt"` (not a string)
Document (Root)	`/`	N/A	N/A	`"bolt washer "`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Depiction of a complete node tree (cont.)

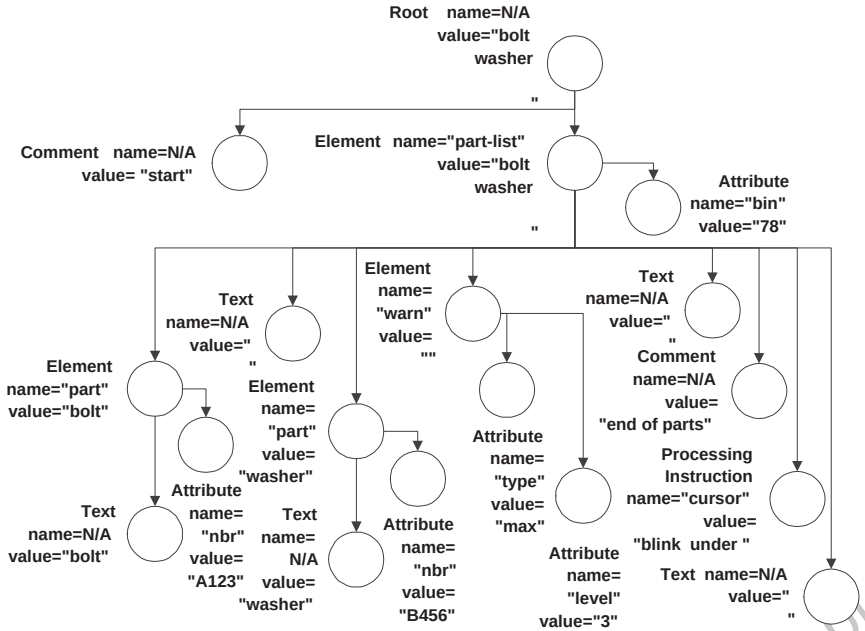
P2 XPath 2 offers a number of other forms of the tests

Node type	Node without specificity	Node in isolation	Isolation example	Example value (inside quotes)
Element	`element()`	`element(`*`elementName`*`)`	`element(part)`	`"bolt"`
Element (validated)	N/A	`schema-element(`*`elementDeclaration`*`)`	N/A	N/A
Attribute	`attribute()`	`attribute(`*`attributeName`*`)`	`attribute(nbr)`	`"A123"`
Attribute (validated)	N/A	`schema-attribute(`*`attributeDeclaration`*`)`	N/A	N/A
Processing Instruction	`processing-instruction()`	`processing- instruction(`*`piTargetName`*`)`	`processing-instruction(cursor)`	`"blink  under "`
Document (Root)	`document-node()`	`document-node(`*`element-test`*`)`	`document-node(element(part-list))`	`"bolt washer "`

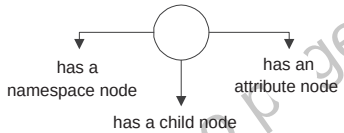
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Depiction of a complete node tree (cont.)

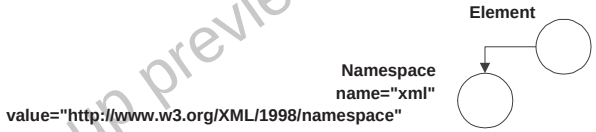
It often helps when designing the transform to sketch out the node tree of the source:



These materials use the following conventions when depicting node trees:



Not depicted is the namespace node for the XML namespace. Such a node is attached to *every* element in the instance:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Depiction of a complete node tree (cont.)

The following is a report of the content of the example on page 94:

```
SHOWTREE - http://www.CraneSoftwrights.com/resources/
1  Comment: {start}
2  Element 'part-list':
2.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
2.A  Attribute 'bin': {78}
2.1  Element 'part' (part-list):
2.1.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
2.1.A  Attribute 'nbr': {A123}
2.1.1  Text (part-list,part): {bolt}
2.2  Text (part-list): {
}
2.3  Element 'part' (part-list):
2.3.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
2.3.A  Attribute 'nbr': {B456}
2.3.1  Text (part-list,part): {washer}
2.4  Element 'warn' (part-list):
2.4.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
2.4.A  Attribute 'level': {1}
2.4.B  Attribute 'type': {max}
2.5  Text (part-list): {
}
2.6  Comment (part-list): {end of parts}
2.7  Proc. Inst. 'cursor' (part-list): {blink  under }
2.8  Text (part-list): {
}
```

Note that the above transform is a free resource available from the given URL and is a useful diagnostic tool to report a processor's view of an arbitrary XML document:

- when a transform does not behave as desired, it can help to see the nodes the processor can see in an XML document in case a visual inspection of that document is misinterpreted by the transform writer

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Depiction of a complete node tree (cont.)

An example including namespace nodes

Consider the well-formed instance `namespace.xml` that contains namespace declarations:

```
01 <?xml version="1.0"?>
02 <a>
03   <b xmlns:x="http://x">
04     <c xmlns:y="http://y">
05       <d x:p="x-p" y:q="y-q" r="r" xmlns="http://z"/>
06     </c>
07   </b>
08 </a>
```

Note in the instance that the `r=` attribute is not in any namespace; it is not in the default namespace that the `d` element is in.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Depiction of a complete node tree (cont.)

The following is a report of the content of the source node tree as generated by a processor, revealing the replication of the namespace nodes:

```
01 SHOWTREE - http://www.CraneSoftwrights.com/resources/
02 1  Element 'a':
03 1.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
04 1.1  Text (a): {
05   }
06 1.2  Element 'b' (a):
07 1.2.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
08 1.2.II  Namespace 'x': {http://x}
09 1.2.1  Text (a,b): {
10     }
11 1.2.2  Element 'c' (a,b):
12 1.2.2.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
13 1.2.2.II  Namespace 'y': {http://y}
14 1.2.2.III  Namespace 'x': {http://x}
15 1.2.2.1  Text (a,b,c): {
16     }
17 1.2.2.2  Element '{http://z}d' (a,b,c):
18 1.2.2.2.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
19 1.2.2.2.II  Namespace: {http://z}
20 1.2.2.2.III  Namespace 'y': {http://y}
21 1.2.2.2.IV  Namespace 'x': {http://x}
22 1.2.2.2.A  Attribute '{http://x}x:p': {x-p}
23 1.2.2.2.B  Attribute '{http://y}y:q': {y-q}
24 1.2.2.2.C  Attribute 'r': {r}
25 1.2.2.3  Text (a,b,c): {
26     }
27 1.2.3  Text (a,b): {
28   }
29 1.3  Text (a): {
30 }
```

Note how there are two namespaces for `b` on lines 7 and 8 and three namespaces for `c` on lines 12 through 14. Every element has the `xml` namespace node and each of `b`, `c`, and `d` have the namespace nodes of all ancestral elements.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Expressions

Chapter 3 - XPath data model
Section 2 - XPath expressions

XPath expressions return items for transformations to act on

- [P2] comment
  - annotating an expression for the human reader
- [P2] simple tuple creation expression
  - basing an expression's return value on the creation of a list of tuples
- [P2] conditional expression
  - basing an expression's return value on an alternation of other values
- data type expressions
  - specifying an expression's value directly or as a calculation

Core data types used in expressions:

- Boolean values
  - the traditional `true` and `false` values used in logical arithmetic
- number values
  - double-precision floating point numbers as defined by IEEE
- string values
  - sequences of UCS characters represented by character codes
- nodes from the source tree (location paths made up of location steps)
  - location paths specify sets of nodes (primarily treated in document order)
  - location steps specify nodes from an axis (primarily treated in proximity order)
  - nodes from the stylesheet can be accessed only if the stylesheet file is opened as a source tree
  - nodes cannot be addressed from the result tree in any way

[P2] XPath 2.0 extensions to XPath 1.0

- W3C Schema data types allowed for atomic values
  - see summary of type hierarchy on page 73
- generalizes both nodes and atomic values as "items"
- introduces the concept of a sequence of items
  - sequences are flat in that there are no embedded sequences within sequences
  - a sequence type (see page 74) is a constraint on the sequence values
  - a data type suffixed by a Kleene cardinality operator
    - no suffix infers one and only one of the item
    - "?" suffix infers zero or one of the item
    - "*" suffix infers zero or more of the item
    - "+" suffix infers one or more of the item

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Expressions (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

[T1] XSLT 1.0 extends XPath 1.0 data types and expressions to include:

- fragments of the result tree
  - packages of result tree nodes representing information that can be added to the result tree on demand
  - one can only copy nodes of a result tree fragment to the result tree or to another result tree fragment
  - write-only during creation
    - one-pass creation of the result tree fragment
  - limited read-only after creation
    - returns the concatenation of all of the text nodes found therein
  - can be copied fully to the result tree after creation
    - any number of times
  - one cannot push or pull nodes of a result tree fragment

[P2] XPath 2.0 extends XPath 1.0 data types and expressions to include:

- arbitrary temporary trees (replaces XSLT 1.0 fragments of the result tree)
  - no concept of "result tree fragment" in XSLT 2.0
  - write-only during creation
    - one-pass creation of the temporary tree
  - fully read-only after creation
    - one can push or pull nodes of a temporary tree just as with a source tree
  - can be copied fully to the result tree after creation
    - any number of times
  - common in two-pass algorithms to create a temporary tree full of intermediate nodes and then push or pull the intermediate nodes in the algorithm to create the final result tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 2 comment expression

P2 Comments in an XPath 2.0 expression:

- delimiters "`(:`" and "`:)`"
- comments are removed from an expression before evaluation and can play no role in the evaluation itself

```
01  <xsl:template match="this (: missing? :)|that (: always there :)">
```

- every XPath expression has to return something and cannot be an empty string after comments have been removed
- the "`()`" expression returns a sequence of no items

Comments can be nested

- but must be properly balanced (a corresponding end sequence after every start sequence)
- unlike XML comments which cannot be nested

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 2 tuple expression

P2 XPath 2.0 includes an operation on a list of tuples of items ("tuple list")

- `for` *`variable-singleton-bindings`* `return` *`result-tuple-expression`*
- a tuple is a set of values treated as a collection
  - one can work on a set or sequence of tuples
- a standalone expression can be a specified operation on a set of tuples comprised of addressed nodes or items
- one variable binding for each member of the tuple
  - the scope of each variable is the following tuple declarations and the return expression
- one result is generated per tuple
  - the result itself may be a sequence of multiple items
- the following returns the sequence "`5, 6, 6, 7, 7, 8`" from a two-tuple of variables
  - `for $x in (1, 2, 3), $y in (4, 5) return $x + $y`
- the following returns the sequence "`5, 4, 6, 6, 7, 8`" from a two-tuple of variables
  - `for $x in (1, 2, 3), $y in (4, $x+2) return $x + $y`
  - latter variable bindings can reference earlier variable bindings
- the following returns a sequence of author strings and book title strings from a one-tuple

```
for $each in distinct-values($books/author)
    return ( $each,
             $books[author=$each]/title/string() )
```

  - the author string is included in the sequence once, and all of their titles follows, followed by the next author
- the following returns a sequence of author nodes and title nodes from a one-tuple

```
for $each in distinct-values($books/author)
    return ( $books[author=$each][1]/author,
             $books[author=$each]/title )
```

  - the author node is included in the sequence once, and all of their title nodes follows, followed by the next author

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 2 tuple expression (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

Very common to create a list of 1-tuples of nodes for repositioning

- `for $n in` *`node-set-address-or-variable`*
  `return` *`sequence-of-nodes-or-values-using-each-$n-singleton`*
  - `$n` represents a new position in the node tree
  - the return sequence of nodes is a set of new positions based on each `$n`

From the new positions a sequence of values can be returned based on each `$n`

- the following returns a sequence of comma-separated names, where the names are created from the given name and surname children of a list of 1-tuples of name nodes
- ```
  string-join( for $contact in $org/contacts/name
                  return concat($contact/first,' ',$contact/last),
               ', ')
  ```
  - the return of the `for` has the same cardinality as the tuple list
  - each string returned in the return sequence is the given name and surname separated by a single space
  - the `string-join()` returns a single string with that returned sequence of strings separated by a comma and space

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 2 conditional expression

Chapter 3 - XPath data model
Section 2 - XPath expressions

P2 XPath 2.0 includes standalone conditional expressions

- an expression can be an if-then-else clause where both sub-clauses are mandatory
- the following returns a number that is either a result of a calculation or a scalar
  - `if ($e/@weight) then $e/@weight * 100. else 100.`
  - the expression gives a full weight to a question without a weight attribute
- a standalone expression cannot be used as an operand in another expression
  - unless it is made standalone using parentheses, as in this example where one might want to invert the weight value calculated above
- the following is *not* allowed:
  - `100. - if ($e/@weight) then $e/@weight * 100. else 100.`
- the following is allowed because the `if` expression is now standalone:
  - `100. - ( if ($e/@weight) then $e/@weight * 100. else 100. )`

Both the `then` and `else` clauses are required

- use "`()`" when no behavior desired
- e.g. `if ( $e/@answer='y' ) then 'Found!' else ()`

Combining addresses, tuples and conditionals makes a complete query

- acting on the conditional result of a tuple expression that includes an address
- the following returns a sequence of numbers
  - ```
    for $each in id('start')//question[@answer='y']
        return if ($each/@weight) then $each/@weight * 100.
                                  else 100.
    ```
  - if there are no questions with an answer attribute with the value "`y`" then the return is the empty sequence and no comparisons are performed
  - otherwise the return is a sequence of numbers, one for each addressed question, being either a calculation or a scalar value
  - suitable as an argument for a function such as `avg()`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath expression types

XPath expressions can be value expressions or address expressions

Value expressions are used for atomic types

- explicit literal values (a.k.a. "primary expressions")
  - e.g. numbers
  - e.g. strings
- explicit calculations
  - e.g. `$base * .85`
  - e.g. `$pretax * (1 + ancestor::order/@tax)`
- evaluating the value of some function
  - e.g.: `sum(`*`sequence-expression`*`)`
    - find all the items specified in the sequence expression
    - convert the string value of each node to a number
    - sum the numbers into a total
    - return the number total as the result of the `sum()` expression evaluation
  - e.g.: `count(`*`sequence-expression`*`)`
    - find all the items specified in the sequence expression
    - return the count of items as the result of the `count()` expression evaluation

Addresses are used for identifying nodes or values derived from nodes

- a "path" expression is set of "step" expressions, separated by "`/`"
  - each step walks around a node tree
  - the rightmost step describes the sequence of nodes or values
  - steps to the left of the rightmost step describe the context of the nodes or values
- the completed evaluation of the expression returns information to be processed
  - a "set of nodes (locations)" is returned for a path ending with a node expression
  - a "sequence of values" is returned for a path ending with a value expression
    - each value is calculated evaluating the expression from the location described by the location path

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath expression types (cont.)

Location path expressions specify sets of nodes for processing or for matching

- selecting a set of nodes from the source tree or XPath 2 temporary tree
  - using an address to identify nodes from the tree to obtain for processing
  - using a location path expression in XSLT
    - e.g.: `<xsl:for-each select="part-name">...</xsl:for-each>`
      - use the given template for each of the `part-name` children
    - e.g.: `<xsl:apply-templates select="part-name/@*"/>`
      - find the templates for all of the attributes of all `part-name` children
    - e.g.: `<xsl:apply-templates select="@part-nbr | @bin"/>`
      - find the template for the attribute `part-nbr` and the attribute `bin` in whatever arbitrary order the XML processor records the attribute in the XPath node tree
    - T2 e.g.: `<xsl:apply-templates select="@part-nbr , @bin"/>`
      - find the template for the attribute `part-nbr` first and then the attribute `bin`, in that order regardless of the order in the XPath data model
  - passing nodes to a function acting on a node set
- specifying a set of matching nodes in XSLT
  - using an address to identify nodes being pushed from the tree to catch for processing
  - using a subset of location path expressions called "patterns"
    - e.g.: `<xsl:template match="@part-nbr">`
      - specify the template for the `part-nbr` attribute of any element
    - e.g.: `<xsl:template match="part-name/@part-nbr">`
      - specify the template for the `part-nbr` attribute with a `part-name` element parent
    - e.g.: `<xsl:template match="@rack | @bin">`
      - specify the template for either the `rack` attribute or the `bin` attribute of any element
  - used wherever nodes are to be matched
    - used in node counting, key collection membership and when choosing between alternative template rules

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Address evaluation context

Chapter 3 - XPath data model
Section 2 - XPath expressions

P1 The "current node list" can contain ***only*** a set of nodes from the source node tree
- all evaluation is done with the focus being a source tree node and nothing else
- a location path address of nodes has no duplicate nodes and all nodes are acted on in document order

P2 The "context list" can contain a sequence of ***any*** information item defined in XPath 2
- evaluation is done with the focus being the context item of any data type
- backwards compatible when using a location path address of nodes (no duplicates and document order)
- an arbitrary sequence may contain duplicates and may address nodes in an arbitrary order with separate location path addresses
  - each address addresses a set of nodes in document order

The context list is a critically important concept in XPath
- the context list is the current "collection" of items being processed, having been specified or addressed for processing
  - the processing of the context list cannot be interrupted
  - if only a subset of items needs to be processed, then only address that subset
- changed in XPath through each step evaluation of a multi-step location path expression
  - addressed and refined as each intermediate result in between two location steps of a location path
  - the result of specifying a node test and filtering by the predicates on each of the nodes from the previous step in the path
  - "*`leftNodes`*`/`*`right`*" is "`for $n in` *`leftNodes`* `return` *`right($n)`*"
- supersedes previous context list until finished being processed
  - when done, current item restores to interrupted position in previous context list
- it is not possible to halt processing of the context list part way
  - all members of the context list are processed in sequence
- exposed in XSLT for the duration of every instruction with a `select=` attribute
  - set throughout the scope for the template in the XSLT instruction
  - `<xsl:apply-templates>` - pushing source tree nodes and, in XSLT 2, temporary tree nodes
  - `<xsl:for-each>` - pulling source tree nodes and, in XSLT 2, atomic values and temporary tree nodes

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Address evaluation context (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

Information available regarding the context list:
- the information changes when an XPath location step or an XSLT instruction selects nodes or atomic values

`last()` returns the current context list size
- the position number of the last context item within the context list
- the value returned does not change for the list because the list size can never change once defined

`position()` returns the current context item's 1-origin position within the context list
- the position number of the current context item within the context list
- the first position is numbered 1, not zero as in many programming languages
- as the processor moves from one item to the next, the position value monotonically increases by one

`.` returns the current context item of the context list
- only one of the items in the context list is current at any time
- the processor moves from one item to the next, starting at the first and ending at the last

XPath exposes all three properties at each step of evaluation
- redefined at each step as the result of the previous step
- the previous step must be a set of nodes for there to be a following step

XSLT exposes all three properties for use within instruction bodies
- the context list is set with every `select=` for the instruction duration
  - end result of the XPath expression's last step's evaluation
  - always delivered to XSLT in document order
- restored to the previous definition of all three values after having processed every member in the list
- T2 no definition of "`.`" at the start of a function
- T2 note that "`/`" is defined only when "`.`" is a node

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Address evaluation context (cont.)

The XPath context list has a number of components:

- context list
  - the collection of items created by a `select=` attribute
- context item
  - a node in the tree
  - [2] an atomic value
- context position
  - a non-zero positive whole number
- context size
  - the length of the context list
  - a non-zero positive whole number greater than or equal to the context position
- variable bindings
  - used when variables are referenced
- function library
  - used when functions are called
- namespace declarations
  - used when namespace prefixes are specified

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location path expression structure

Location path expressions address nodes by the relationships of the nodes to each other

- first step in the location path is significant
  - arbitrary location from well-defined position
    - positions usable in pattern expressions
      - begins with a function returning a node-set from a source node tree
        - `id()`
        - `key()`
    - positions not usable in pattern expressions
      - begins with any other function returning a node-set
        - *`function`*`()`
      - begins with a node-set variable
        - `$`*`node-set-variable-qname`*
      - begins with a parenthesized union or sequence expression
        - `(` *`node-set-expression`* `)`
        - `(` *`node-set-expression`* `|` *`node-set-expression`* `)`
        - [2] `(` *`node-set-expression`* `,` *`node-set-expression`* `)`
  - absolute location from root of tree (document node)
    - begins with "`/`"
    - may be followed with a relative location expression
    - e.g. `/a/b`
      - when selecting: "the `b` children of the `a` document element"
      - when matching: "any `b` child of the `a` document element"
    - if "`.`" is a node, then "`/`" is the top of that node's tree
    - no definition of "`/`" when "`.`" is undefined or is an atomic item
    - the tree being accessed is an important nuance when dealing with multiple source node trees
      - the transform can store the root of the initial source node tree in a top-level variable in order to always have access to the starting tree
  - relative location from current position
    - does not begin with "`/`"
    - node relationships evaluated relative to the exposed current context item
    - e.g. `a/b`
      - when selecting: "the `b` children of the `a` children of the current node"
      - when matching: "any `b` child of any `a` element"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location path expression structure (cont.)

Location path expressions (cont.)

- multiple steps in a path are separated using "`/`" after steps returning nodes
  - "*`leftNodes`*`/`*`right`*" is "`for $n in `*`leftNodes`*` return `*`right($n)`*"
    - a "`/`" can only follow an expression returning nodes
    - "`/`" returns the union of evaluating each node on the left-hand side with the result of the expression on the right-hand side with each node
    - the last step can return a value of any data type
    - "`a/b/c`" evaluates as "`(a/b)/c`"
  - steps in a location path used in a selection are evaluated left to right
    - one step followed by the next step to the right
    - each step refines the node-set based on specified criteria
    - the nodes being selected are the right-most (last) nodes in the path
  - steps in a location path used in a match pattern are evaluated right to left
    - one step in the context of the next step to the left
    - each step narrows the context in the ancestry
    - the node being matched is the right-most (first) node in the path
  - right-most step is what is being addressed
    - steps to the left are only the context of the right-most step
  - recall examples on page 107
  - a location path other than "`/`" cannot end with "`/`"
    - every "`/`" must be followed with nodes or values (not a mixture)
    - consider `max(step/(@repair*1.25,+@replace)[1])`
      - the "`+@replace`" atomizes the attribute node into an atomic value
      - the "`[1]`" ensures `@replace` is used only when `@repair` is absent
- P2 very flexible combinations of steps available for selecting (not matching)
  - because any expression can be used in any step
  - e.g. "`a/(b,c)/d`" evaluates as "`a/b/d,a/c/d`"
  - e.g. "`a/*/name(.)`"
    - applies the name function to each child node of child `a`, returning a sequence of strings
  - e.g. "`a/*/@idref/id(.,$tree)/name(.)`"
    - applies the name function to each node returned by passing the `idref` attribute of each child node of child `a` to `id()`, returning a sequence of strings
    - note this example is no different than writing "`id(a/*/@idref)/name(.)`"
- it is not an error to ask for something that isn't there
  - P1 a syntactically valid expression that addresses a node that does not exist returns the empty set
  - P2 a schema-aware transformation can detect misspelled names and names in unexpected contexts when the context is known

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location path expression structure (cont.)

Location path expressions (cont.)

- a single expression can be the union of multiple node expressions
  - complete location expressions separated with "`|`", e.g. "`$a | $b`"
    - logical `or` for union of node-sets (not logical `and` for intersection)
    - XSLT behavior "feels" different based on where the operator is being used
      - acts like the natural language "or" (choice) when performing pattern matching
        - `match="a | b"` matches element nodes named "`a`" or element nodes named "`b`"
      - acts like the natural language "and" (aggregate) when performing node selection for evaluation
        - `select="a | b"` selects child element nodes named "`a`" and child element nodes named "`b`"
          - any duplicate nodes addressed by the two expressions are reduced to a single copy
          - the items are acted on in document order
    - P2 the "`union`" keyword is allowed as in "`$a union $b`"
    - described in detail on slide 307
- P2 a single expression can be the intersection of multiple node expressions
  - complete location expressions separated with "`intersect`"
  - "`$a intersect $b`" returns those nodes in `$a` that are also in `$b`
  - not allowed in a `match=` attribute
- P2 a single expression can be the difference of multiple node expressions
  - complete location expressions separated with "`except`"
  - "`$a except $b`" returns those nodes in `$a` that are not in `$b`
  - not allowed in a `match=` attribute
- P2 a single expression can be the sequence of multiple item expressions
  - not allowed in a `match=` attribute
  - complete location expressions separated with "`,`"
    - arguments evaluated and returned in sequence order as in "`$a , $b`"
      - any duplicate nodes addressed by the two operands are acted on as many times as found in sequence
        - any duplicates found in any one of the operands in the expression are ignored
      - the items are acted on in sequence order
  - sequences expressed as being nested are flattened
    - "`(p,(q,r),s)`" is interpreted as "`(p,q,r,s)`"
    - nested empty sequences are removed
    - "`((),p)`" is interpreted as "`(p)`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location steps

A step can be either a primary-based step or an axis-based step

- primary-based step is an arbitrary expression
  - may act on the context item regardless of its data type
  - P1 returns zero or more source tree nodes
  - P2 returns zero or more items of any data type
- axis-based step is an address in a tree of nodes
  - acts on the context item which must be a tree node
  - returns zero or more tree nodes resulting from the step
  - P1 tree nodes must be source tree nodes
  - P2 tree nodes may be source tree nodes or temporary tree nodes

Second and subsequent steps allowed only if left-hand side of "`/`" returns tree nodes

- the step is evaluated repeatedly, one node at a time, for each node on the left-hand side, evaluating the right-hand side expression with that node treated as the current node
- "*`leftNodes`*`/`*`right`*" is "`for $n in `*`leftNodes`*` return `*`right($n)`*"
- step result is the union of step evaluations for all left-hand-side nodes

P1 Second and subsequent steps must be an axis-based step

- a source tree node is always the input to the step
  - result tree fragment nodes are not allowed as input to the step
- such steps can only produce source tree nodes as the output of the step

P2 Any step can be a primary-based step or an axis-based step

- primary-based steps other than the last step must return nodes
  - the last step can return nodes or atomic values
- whatever item output from the previous step is input to the current step
- the current step can produce any sequence of data type output as input to the next step
- powerful opportunity to map function calls against source tree nodes in a single expression

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location steps (cont.)

Each primary-based location step in a location path is comprised of:

`primary-expression` **`[`*`predicate`*`][`*`predicate`*`]`**

Primary expressions include:

- variable reference
  - e.g. `$runningValues`
- parenthesized expression
  - e.g. `( $runningValues | * )`
    - this is the union of the nodes in `$runningValues` with the element children of the current node
    - any duplicates are removed and the result set is treated in document order
- function call
  - e.g. `id(@linkends)`
    - find referenced elements in the current tree
  - P2 e.g. `xs:float(123)`
    - casting the double value `123` into a float value
  - P2 e.g. `xs:date(schedule/startDate)`
    - casting the text value of the element into a date value

P2 Primary expressions also include only a few data types (see page 73):

- string literal
  - e.g. `"12340000"` or `'12340000'`
    - a string of characters, not a number
- integer literal (digits without a decimal point "`.`")
  - e.g. `12340000`
    - a whole number without any part
- decimal literal (digits with a single decimal point "`.`")
  - e.g. `12340000.`
    - a whole number that happens not to have a part, but it may have a part if desired
    - this example could be cast to an integer without error, but if it had a part that would not be true
- double literal (decimal literal followed by "E-notation" for powers of 10)
  - e.g. `12.34E6`
  - the same as `12340000.` but with greater precision
- to get values of other types (W3C schema types or user types) use the casting function call

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

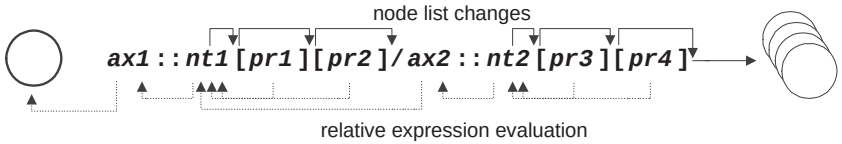
Location steps (cont.)

Each axis-based location step in a location path is comprised of:

***axis*`::`*nodetest* `[`*predicate*`][`*predicate*`]`**

- an axis identifier
  - indicating the direction from current node
- a node test
  - filtering nodes by type and name
- zero or more predicates
  - filtering nodes by qualifying expressions tested against each node

The current node list changes at and during every step evaluation



- changes at every node test by being set to the nodes being addressed along the axis
- changes after every predicate by eliminating the members that test `false`
- note in the diagram how the dotted lines indicate the relative addressing of the predicates is to the nodes tested, and the nodes are relative to the nodes tested in the previous step
- note in the diagram how the solid lines indicate the setting of the node list to the nodes being tested and that the node list changes with the application of each predicate
- the resulting node list contains the nodes of the type of the rightmost node test

For example, the following single location path has 4 steps:

- ```
  id(parent::*/@idref)/ancestor-or-self::frame
                      /descendant::fig[caption]
                      /@image
  ```
- recall the source tree has many parent, child and sibling relationships (page 72)
- arbitrary white space can be added to try to make location paths more legible
- the first step jumps to an arbitrary location based on the unique identifier of an element being equal to the `idref=` attribute of the parent of the current node
- the second step moves up the hierarchy, if necessary, to the `frame` element
- the third step uses a predicate to filter from all descendent `fig` elements only those with child `caption` elements
- the fourth step moves to the `image` attribute of each of the `fig` elements selected
- note that the expression uses some abbreviations described later in this chapter

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Axes

The use of a double colon "`::`" immediately follows the use of an axis name.

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate*`][`*predicate*`]`**

Two attachment axis directions from the current node, each in arbitrary order

- each processor can have a different order from any other processor
- `attribute::`
  - attribute nodes (specified in element or defaulted in DTD)
- `namespace::`
  - namespace nodes (specified in element or ancestral element)

P2 Namespace axis is deprecated in XPath 2.0

- no obligation for an implementation of XPath 2.0 to support the namespace axis
- a challenge for some information processing tasks where namespace management is critical to the algorithm
- many developers will probably support this axis

P2 Namespace information available using XPath 2.0 functions that are not deprecated

- `in-scope-prefixes()` and `namespace-uri-for-prefix()`
- e.g. replacing `name(namespace::*[.=$nsuri])`
- with:
- `in-scope-prefixes($node)[namespace-uri-for-prefix(.,$node)=$nsuri]`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
  ```

Axes (cont.)

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate* `][`*predicate* `]`**

Eleven parent/child axis directions from current node, each in proximity order

- proximity order counts from the closest node as 1 and other nodes counting away from current node
  - document order for those nodes that start after the start of the current node
  - reverse document order for those nodes that start before the start of the current node
- a node tree depiction is included at the beginning of this chapter at The need for abstractions (page 72)
  - letters in examples below track the order of the nodes in the example for each axis
- `self::`
  - the current node (`J`), shown as the thick-lined node
- `child::`
  - immediate child nodes (`K`,`L`,`O`,`P`,`Q`)
- `descendant::`
  - all descendent nodes (`K`,`L`,`M`,`N`,`O`,`P`,`Q`,`R`,`S`)
- `descendant-or-self::`
  - the current node and its descendent nodes (`J`,`K`,`L`,`M`,`N`,`O`,`P`,`Q`,`R`,`S`)
- `parent::`
  - the parent node (`F`)
  - this is the attaching element node for attached attribute and namespace nodes
- `ancestor::`
  - the parent and its ancestors (`F`,`A`,`root`)
- `ancestor-or-self::`
  - the current node and its ancestors (`J`,`F`,`A`,`root`)
- `preceding-sibling::`
  - all preceding nodes that are siblings of the current node (`I`,`G`)
- `following-sibling::`
  - all following nodes that are siblings of the current node (`T`,`U`)
- `preceding::`
  - all nodes wholly contained before the start of the current node (`I`,`H`,`G`,`E`,`D`,`C`,`B`)
- `following::`
  - all nodes wholly contained after the end of the current node (`T`,`U`,`V`,`W`,`X`,`Y`,`Z`)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node tests

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate* `][`*predicate* `]`**

The node test specifies what type and optionally which name of node to look for along the direction of the axis specified or implied:

- a node by its node type along any axis
  - `processing-instruction()`
    - a processing instruction node (regardless of name)
  - `comment()`
    - a comment node
  - `text()`
    - a text node
  - `node()`
    - a node of any type (the wildcard)
  - note that even though the above can be used on any axis, the attached axes will never have processing-instruction, comment or text nodes
- P2 a node by its node type along any axis
  - `document-node()`
    - a document node (regardless of the document element)
  - `element()`
    - an element node (regardless of its name or type)
  - `attribute()`
    - an attribute node (regardless of its name or type)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node tests (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate*`][`*predicate*`]`**

Node tests based on the node name:

- a name: `processing-instruction(`*name-string-expression*`)`
  - a processing instruction node by its name
  - P2 `processing-instruction(`*name*`)`
- a name: *name*
  - when using the namespace axis: a namespace node
  - when using the attribute axis: an attribute node not in any namespace
  - otherwise: an element node not in any namespace
  - note that "not in any namespace" also means not in the default namespace
- a namespace-qualified name: *prefix*`:`*name*
  - when using the namespace axis: empty node-set
  - when using the attribute axis: an attribute node in the given prefix's namespace
  - otherwise: an element node in the given prefix's namespace
- the wildcard: `*`
  - when using the namespace axis: all namespace nodes
  - when using the attribute axis: all attribute nodes
  - otherwise: all element nodes in any namespace (including the default namespace)
- the namespace-qualified wildcard: *prefix*`:*`
  - when using the namespace axis: empty node-set
  - when using the attribute axis: all attribute nodes in the given prefix's namespace
  - otherwise: all element nodes in the given prefix's namespace
- P2 the wildcard namespace: `*:`*name*
  - when using the namespace axis: empty node-set
  - when using the attribute axis: all attribute nodes in the given prefix's namespace
  - otherwise: all element nodes in any namespace with the given name

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node tests (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate*`][`*predicate*`]`**

P2 More node tests available in XPath 2.0:

- based on name:
  - `element(` *element-name-or-wildcard* `)`
    - all elements along the axis that match the name or wildcard
  - `attribute(` *attribute-name-or-wildcard* `)`
    - all attributes along the attribute axis that match the name or wildcard
    - using this explicitly with any axis other than the attribute axis is meaningless
  - `document-node( element(` *element-name-or-wildcard* `) )`
    - all document nodes along the axis with the given matching document element
  - the wildcard can be a "`*`" argument or an absent argument
- based on specified named type:
  - `element(` *element-name-or-wildcard*`,` *schema-type* `)`
    - all elements along the axis that match the named schema type (or a type derived from this type) from the currently imported schema
  - `attribute(` *attribute-name-or-wildcard*`,` *schema-type* `)`
    - all attributes along the axis that match the named schema type (or a type derived from this type) from the currently imported schema
    - using this explicitly with any axis other than the attribute axis is meaningless
  - `document-node( element(` *element-name-or-wildcard*`,` *schema-type* `) )`
    - all document nodes along the axis with the document element that matches the named schema type (or a type derived from this type)
  - the wildcard is "`*`"
- based on schema-declared type:
  - based on the type as found in the schema declaration for the item
  - `schema-element(` *element-name* `)`
    - all elements along the axis that match both the name and the schema type for that named element
  - `schema-attribute(` *attribute-name* `)`
    - all attributes along the axis that match both the name and the schema type for that named attribute
    - using this explicitly with any axis other than the attribute axis is meaningless
  - `document-node( schema-element(` *element-name* `) )`
    - all document nodes along the axis with the document element that matches both the name and the schema type for that named element

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node tests (cont.)

The "XML default namespace" is for un-prefixed elements: `<para xmlns="..uri..">`

- un-prefixed attribute names are always in no namespace

The "XPath default namespace" is that used for un-prefixed elements in XPath expressions as `b` is in `/x:a/b/y:c`

- in the example, `a` is in the namespace associated with the prefix `x` and `c` is in the namespace associated with the prefix `y`
- the XML default namespace and the XPath default namespace are always distinct and are independently set
  - the XPath default namespace is always the null namespace in XSLT 1
  - use `xpath-default-namespace=` in XSLT 2
  - changes to the XML default namespace do not impact on the XPath default namespace (and vice versa)

Very common source of XSLT stylesheet errors

- for example, XHTML documents utilize the default namespace
- XSLT 1 stylesheets require every match and every select to use prefix-qualified names in XPath addresses
- XSLT 2 stylesheets can conveniently use `xpath-default-namespace=`
- the following will *not* match the element `{http://www.w3.org/1999/xhtml}b`

```
01 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
02                 version="1.0">
03
04 <xsl:template match="b" xmlns="http://www.w3.org/1999/xhtml">
05  ...template for bold handling with bad match expression...
06 </xsl:template>
07
08 </xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Node tests (cont.)

Chapter 3 - XPath data model
Section 2 - XPath expressions

Consider an element in the source file using the name `{urn:X-b}a` in the default namespace:

```
01 <a xmlns="urn:X-b">This is a!</a>
```

The following XSLT 1 stylesheet does not find the XML element at all:

- the XML default namespace declaration on line 6 is ignored by XPath
- `out1` is in no namespace and `out2` is in the "`urn:X-b`" namespace

```
01 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
02                 version="1.0">
03 <xsl:template match="/">
04   <out1>
05     <xsl:value-of select="string(a)"/>
06     <out2 xmlns="urn:X-b">
07       <xsl:value-of select="string(a)"/>
08     </out2>
09   </out1>
10 </xsl:template>
11 </xsl:stylesheet>
```

The following XSLT 2 stylesheet finds the element only using the address on line 7, not on line 5

- the address on line 5 is for "`a`" in the "`urn:X-z`"
- `out1` is in no namespace and `out2` is in the "`urn:X-z`" namespace

```
01 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
02                 version="2.0" xpath-default-namespace="urn:X-z">
03 <xsl:template match="/">
04   <out1>
05     <xsl:value-of select="string(a)"/>
06     <out2 xmlns="urn:X-z" xsl:xpath-default-namespace="urn:X-b">
07       <xsl:value-of select="string(a)"/>
08     </out2>
09   </out1>
10 </xsl:template>
11 </xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Abbreviations

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate* `][`*predicate* `]`**

Several abbreviations can make writing expressions more succinct

- some abbreviations are only an axis specification
  - requires a node test
  - allows predicates to be used within the step
  - omitted axis specification
    - [2] a special case for XPath 2.0 that the `attribute()` and `schema-attribute()` tests address nodes along the `attribute::` axis
    - all other node tests test address nodes along `child::` axis
  - the character '`@`'
    - nodes along the `attribute::` axis according to the node test and possible predicate
- other abbreviations are complete location steps
  - prohibits the direct use of predicates within the step
  - the character "`.`"
    - `self::node()`
      - the current node regardless of the node type when the context is a node
    - the current item when the context is not a node
    - can be used stand-alone in the expression
  - the sequence "`..`"
    - `parent::node()`
      - the parent node if the current node is not an attached node
        - element, text, processing-instruction or comment
      - the attaching node if the current node is an attached node
        - attribute or namespace
      - the empty node-set if the current node is the root node
    - can be used standalone in the expression
  - the sequence "`//`"
    - `/descendant-or-self::node()/`
    - abbreviated absolute location path when used at the start of an expression
    - abbreviated relative location path when used elsewhere in an expression
    - must be followed by another location step in the expression
- [2] predicates are allowed on the following abbreviated steps in XPath 2.0
  - `.[`*predicate*`][`*predicate*`]`
  - `..[`*predicate*`][`*predicate*`]`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Predicates

Recall the components in the syntax of an axis-based location step:

***axis*`::`*nodetest* `[`*predicate* `][`*predicate* `]`**

A predicate filters a set or sequence, keeping items by applying a test to each member:

- the qualifying expression keeps some items in and filters other items out
  - the data type of the expression determines the effective Boolean value
  - a value of `false` causes the node to be removed from the step's evaluation
- [2] specifiable also on node sequences, atomic sequences and primary steps
- specified within square brackets "`[`" and "`]`"
  - [2] only singleton sequences or sequences starting with a non-empty node set
  - numeric expression value: `question[3]` or `question[last()]`
    - ordinal position (1-origin) in the current node list
      - equivalent to `child::question[position()=3]` and `child::question[position()=last()]` respectively
      - addressing "the third element child named `question`" and "the last element child named `question`" respectively
    - the predicate is `true` for only the node with the given ordinal position
  - node-set expression value: `question[@answer]` or `question[guess]`
    - node presence test
      - equivalent to `child::question[attribute::answer]` and `child::question[child::guess]` respectively
      - addressing "all `question` children with an `answer` attribute" and "all `question` children with `guess` children" respectively
    - the predicate is `true` only if the selection of nodes specified returns a non-empty set of nodes
  - string expression value: `question[string(.)]`
    - non-empty string test
      - equivalent to `child::question[string(.)!='']`
      - addressing "all `question` children whose element string value is the empty string"
    - the predicate is `true` only if the string calculated is not the empty string
  - other expression value: `question[count(guess)=5]`
    - converted to Boolean according to `boolean()`
      - addressing "all `question` children with five `guess` children"
    - the predicate is `true` only if the expression evaluates to `true`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Predicates (cont.)

- for numeric predicates, the node ordering depends on the source of the node-set
  - nodes specified in an *axis* expression are treated in proximity order
    - nodes selected along the reverse axes (towards the start of the document) are counted in reverse document order and nodes selected along forward axes (towards the end of the document) are counted in document order
    - when the axis includes the context node, the context node is at position one and the node in the set closest to the context node is at position two, and so on through the list
    - when the axis does not include the context node, the node in the set closest to the context node is at position one, the next closest is at position two, and so on through the list
  - nodes returned in an XPath intermediate *step* are treated in document order
    - function return node sets and "`()`" sets
  - nodes returned by a location path *set* expression are in no XPath order
    - XSLT treats returned node sets in document order
      - confusingly as "filters the node-set with respect to the child axis"
  - `preceding-sibling::*[1]` does not necessarily select the same node as `(preceding-sibling::*)[1]`
    - the first is a complete location step expression while the second is a predicate applied to a location path expression
- multiple predicates are cumulative
  - evaluated left to right in the syntax
  - specified in adjacent predicate expressions
  - applied in turn to each node in the resulting set from applying the previous predicate
  - number predicates are calculated on nodes remaining from a previous predicate
    - e.g. `task[@tools][1]`
      - the first task that makes reference to tools
      - not necessarily the first task of the task's parent, because the first task might not make reference to tools
- all predicates are applied to each node in current node list or set
  - the predicate is evaluated in the context of each node in turn
    - the predicate expression temporarily changes the context of evaluation to produce the result
    - the resulting context at the end of a given predicate's evaluation is not material
  - the node stays in set if the predicate's result Boolean value is `true`
  - otherwise, the node is removed from the list

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Example node-set expressions

Some example location path addresses written as a `select=` attribute in XSLT:

1. `self::partNbr`
   - the current node if it is an element named "`partNbr`", if not, an empty node-set
2. `child::partNbr`
   - child element nodes from the current node with the name "`partNbr`"
3. `partNbr`
   - child element nodes from the current node with the name "`partNbr`"
4. P2 `element(partNbr)`
   - child element nodes from the current node with the name "`partNbr`"
5. `./partNbr`
   - child element nodes from the current node with the name "`partNbr`"
6. `*[self::partNbr]`
   - child element nodes from the current node with the name "`partNbr`"
7. `.//partNbr`
   - descendent element nodes from the current node with the name "`partNbr`"
8. `//partNbr`
   - descendent element nodes from the root with the name "`partNbr`" (i.e. all such element nodes in the entire document)
9. `../partNbr`
   - sibling (and possibly self) element nodes with the name "`partNbr`" by first going to the parent with "`..`" and then to the parent's children
10. `@type`
    - attached attribute nodes with the name "`type`"
11. P2 `attribute(type)`
    - attached attribute nodes with the name "`type`"
12. `../@type`
    - the parent's attached attribute nodes with the name "`type`"
13. `../partNbr/@type`
    - the attribute nodes with the name "`type`" of the sibling element nodes with the name "`partNbr`"
14. `partNbr/@type`
    - the attribute nodes named "`type`" of the child element nodes with the name "`partNbr`"
15. `/..`
    - a guaranteed empty node-set (the root never has a parent node)
16. P2 `()`
    - an empty sequence

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Example node-set expressions (cont.)

Example expressions incorporating predicates:

1. `partNbr[@type]`
   - the child "`partNbr`" element nodes that have the attribute named "`type`"
2. `partNbr[@type='cots']/@price`
   - the attribute node named "`price`" of the child element nodes with the name "`partNbr`" that have the attribute named "`type`" equal to the string '`cots`'
3. `partNbr[position()=3]`
   - the third child element node named "`partNbr`"
4. `partNbr[3] | partNbr[5]`
   - the third and fifth child element nodes named "`partNbr`"
   - because the data type of the predicates is a number, this is equivalent to:
     - `partNbr[position()=3] | partNbr[position()=5]`
5. `partNbr[3 or 5]`
   - this returns all of the children named "`partNbr`" because the data type of the predicate is Boolean
   - "`3 or 5`" reduces to "`true() or true()`" which is always `true()`
6. `partNbr[position()>=3 and position()<=5]`
   - the third through fifth child element nodes named "`partNbr`"
   - remember that "<" must be escaped in XML attributes
7. `partNbr[@type][3]`
   - the third child "`partNbr`" element node named that has an attribute named "`type`"
8. `partNbr[3][@type]`
   - the third child "`partNbr`" element node *if* that node has an attached attribute named "`type`", otherwise, it returns the empty set
9. `(//partNbr)[1]`
   - the first element node with the name "`partNbr`" in the entire document
10. `//partNbr[1]`
    - descendent element nodes from the root with the name "`partNbr`" that are the first amongst sibling `partNbr` elements
11. `*[not(self::partNbr | self::partList)]`
    - all children elements except elements named "`partNbr`" or "`partList`"
12. P2 `* except ( partNbr | partList )`
    - all children elements except elements named "`partNbr`" or "`partList`"
13. P2 `*/* except ( partNbr | partList )`
    - all grandchildren elements (the "`except`" doesn't address any grandchildren, so there is no overlap between the two operands)
14. P2 `*/* except ( */partNbr | */partList )`
    - all grandchildren elements except those named "`partNbr`" or "`partList`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Example node-set expressions (cont.)

Example expressions that work in the reverse document order, thus could not be patterns in XSLT match attributes:

1. `following-sibling::text()[1]`
   - the closest following text node that is a sibling
2. `following-sibling::text[1]`
   - the closest following element node named "`text`" that is a sibling
3. `following::partNbr[1]`
   - the closest following element node with the name "`partNbr`"
4. `following::text()`
   - all text nodes that follow the current node (does not include descendants)
5. `following::partNbr[1]/following::text()`
   - all text nodes that follow the closest following element node with the name "`partNbr`"
6. `ancestor-or-self::*/@xml:lang`
   - the attributes named "`xml:lang`" of self and all the ancestral element nodes of any name
7. `ancestor-or-self::*[@xml:lang][1]`
   - self or the closest ancestral element node of any name that has an attribute named "`xml:lang`"
8. `ancestor-or-self::*[1][@xml:lang]`
   - self or the closest ancestral element node of any name if and only if that element has an attribute named "`xml:lang`" (if the closest does not have the attribute, the result is the empty set of nodes)
9. `ancestor-or-self::*[@xml:lang][1]/@xml:lang`
   - the attribute named "`xml:lang`" of self or the closest ancestral element node of any name that has an attribute named "`xml:lang`"
10. `ancestor-or-self::*/@xml:lang[1]`
    - the attributes named "`xml:lang`" of self and all the ancestral element nodes of any name (the predicate has no effect)
11. `(ancestor-or-self::*/@xml:lang)[1]`
    - the attribute named "`xml:lang`" of the furthest ancestral element node or self of any name that has an attribute named "`xml:lang`" (the set is in document order)
12. `(ancestor-or-self::*/@xml:lang)[last()]`
    - the attribute named "`xml:lang`" of self or the closest ancestral element node of any name that has an attribute named "`xml:lang`" (the set is in document order)
13. `id(@reference)/@type`
    - the attribute named "`type`" of the element nodes returned by calling the `id()` function with the value of the "`reference`" attribute attached to the current node

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location path expression evaluation summary

XPath location path expression evaluation begins at the first step:

- an arbitrary location
  - oriented to the node or nodes resulting from executing a function or the reference to a node-set variable
- an absolute location
  - oriented to the root node of the tree of the current node
  - only if the context item is a node
  - only if the XQuery process is initialized with a source tree
- a relative location
  - oriented to the current node
  - only if the context item is a node
  - if the XQuery process is initialized with a source tree this returns the root node when not used in an XPath address

Each step is evaluated:

- resulting node list of previous step is the current node list for given step
- step conditions evaluated for each member of node list
  - each evaluation uses each member as current node
  - each evaluation returns a set of nodes
- resulting step node list is union of all evaluations in the step

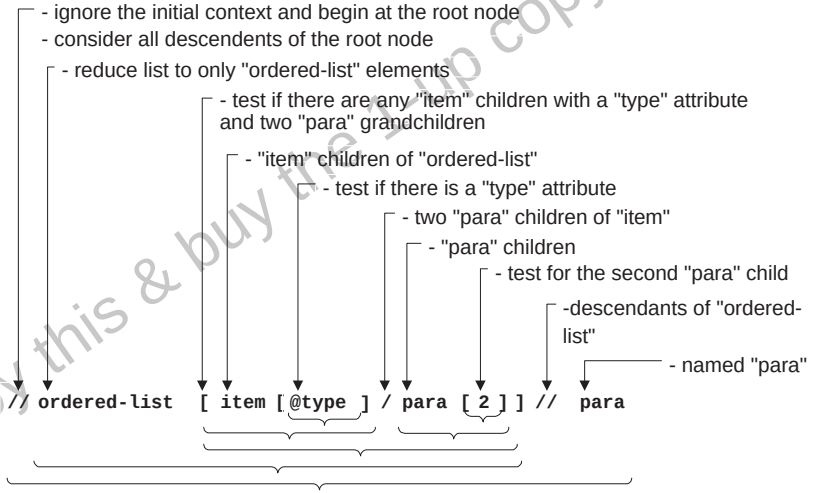
Evaluation continues:

- steps evaluated left-to-right
- evaluation terminates when current node list empty
  - resulting node list for the entire expression is the empty node list
- the final step's result becomes the expression's result
  - the current node list after the last step is evaluated

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Location path expression evaluation summary (cont.)

Consider the stepwise evaluation of the following expression utilizing predicates:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing of node-sets from reverse axes

Important nuances regarding the processing of nodes selected by axes are as follows:

- recall that predicates are oriented by proximity to the current node, thus predicates applied to nodes found on reverse axes act on the node-sets in reverse document order, while predicates applied to nodes found on forward axes act on the node-sets in document order
- when the resulting set selected by the given location step is processed *as a collection of nodes*, it is processed in document order, not in axis order
  - when a predicate is applied to a node-set expression containing nodes selected by axes
  - in an intermediate step for a following location step expression

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing of node-sets from reverse axes (cont.)

Consider the XML file `nodeset.xml`:

```
<?xml version="1.0"?>
<set>
<item>A</item>
<item>B</item>
<item>C</item>
</set>
```

Processed with the XSLT file `nodeset.xsl`:

```
<?xml version="1.0"?><!--nodeset.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;">]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="//item[last()]"> <!--work from the end-->
    <xsl:text>"preceding-sibling::item[1]": </xsl:text>
    <xsl:value-of select="preceding-sibling::item[1]"/>
    <xsl:text>&nl;"(preceding-sibling::item)[1]": </xsl:text>
    <xsl:value-of select="(preceding-sibling::item)[1]"/>
    <xsl:text>&nl;for each "preceding-sibling::item":</xsl:text>
    <xsl:for-each select="preceding-sibling::item/text()">
      <xsl:text>&nl;  Item: </xsl:text><xsl:value-of select="."/>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Produces the following result:

```
"preceding-sibling::item[1]": B
"(preceding-sibling::item)[1]": A
for each "preceding-sibling::item":
  Item: A
  Item: B
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Mimicking an XPath 1.0 conditional expression for node sets

Chapter 3 - XPath data model
Section 2 - XPath expressions

P2 XPath 2.0 allows conditional expressions

- any Boolean test can return an alternative between two possible specified true/false values

T1 XPath 1.0 does not include a conditional expression operator.

- Sometimes one has to choose between nodes depending on the existence of one of the nodes.
- There is no operator to choose between two nodes in a single XPath expression.

Consider the following choice between a base name child element value and its alternate sort name child element value for sorting as might be used in a Topic Maps application (note the XSLT conditional is described in "If - Else If - Else" conditionality (page 140)):

```
<xsl:for-each select="//name">
  <xsl:choose>
    <xsl:when test="sortname">
      <xsl:value-of select="sortname"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="basename"/>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:text>&nl;</xsl:text>
</xsl:for-each>
```

The above successfully prioritizes the use of the sort name above the use of the base name:

- If the sort name does not exist, the base name is used.
- If the sort name does exist, the sort name is used.

Because the union expression is not in error for an absent operand, one can choose only one member from the result of the union:

```
<xsl:for-each select="//name">
  <xsl:value-of select="(basename|sortname)[last()]"/>
  <xsl:text>&nl;</xsl:text>
</xsl:for-each>
```

This technique requires knowledge of the document order of the two nodes.

- Use the predicate `[last()]` to choose the last in document order
- Use the predicate `[1]` to choose the first in document order

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 4 - Processing model

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

A predictable behavior for processors

The basic processing model for XSLT is designed to ensure predictability

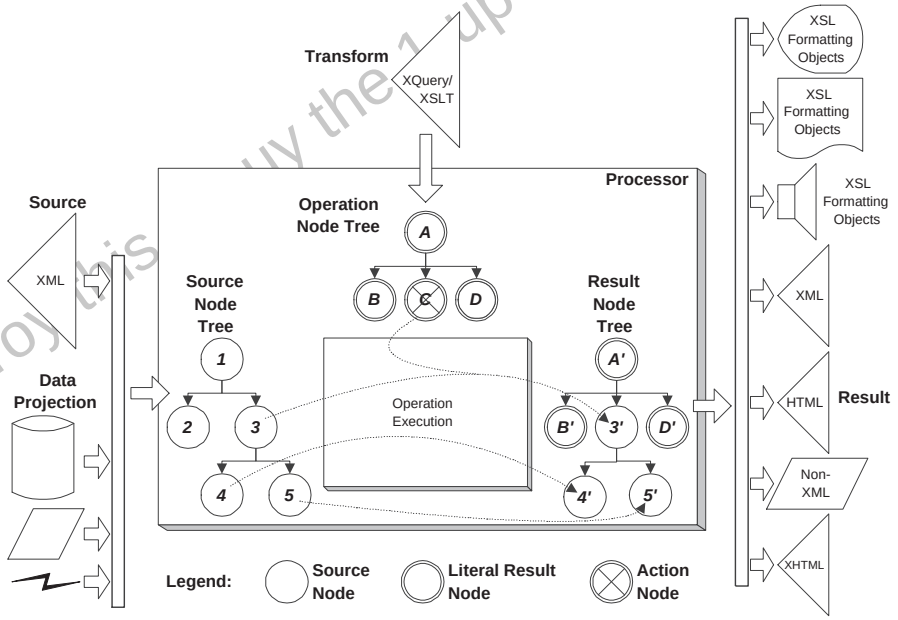
- predictable processing behavior every time
  - all aspects of the processing are well-defined
- processor builds the operation tree of nodes from the transformation expression
  - some nodes of which are evaluations and calculations
  - some nodes of which may be engaging extensions implemented by the processor
  - the remainder of which are literal result elements that are to be used in the construction of the result
- processor builds the source tree of nodes from the primary source resource
  - a primary source tree is not required
  - the markup of the XML source document is not material
  - the vocabulary used in the source is not material to the processor
    - with the exception of `xml:*=` attributes available for use with all XML vocabularies
- result tree construction starts with operation tree
  - start with the template of the template rule for the root node
  - T2 alternatively start at a specified named template or mode
- the transform constructs the content of the result node tree in result parse order in one pass
  - the transform writer must plan the flow of the transform process according to the document order of the result
  - other source trees are created from other source files on request by the transform
  - components from the source trees are obtained where required when executing instructions found in the transform
  - once a portion of the result tree is completely generated there is no method of returning to modify the result tree in any way
- the result tree of nodes may be serialized into markup
  - XML markup
  - HTML markup (using SGML lexical conventions)
  - XHTML markup (using XML lexical conventions)
  - simple text
  - syntax and lexical conventions recognized by the particular implementation of the processor (binary or text)
  - interpreted XSL formatting objects (e.g.: display, print, aural, etc.)
  - remember the processor is *not* required to support any particular serialization method and may choose to serialize the tree as XML only if at all

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

A predictable behavior for processors (cont.)

A simple illustration of the basic process

- the illustrated operation node tree has literal result nodes and a single operation node
- the operation node obtains information from a particular point in the source tree
- not shown in this illustration are built in behaviors copying nodes to the result tree



The diagram's action nodes are created from XSLT instructions.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

A predictable behavior for processors (cont.)

The XSLT instructions covered in this chapter are as follows.

Instructions related to process control:

- `<xsl:if>`
  - single-state conditional inclusion of a template
- `<xsl:choose>`
  - multiple-state conditional inclusion of one of a number of templates
- `<xsl:when>`
  - single-state conditional inclusion of a template within a multiple-state condition
- `<xsl:otherwise>`
  - default-state conditional inclusion of a template within a multiple-state condition

Instructions to pull information from the source tree or to calculate values:

- `<xsl:copy-of>`
  - add to the result tree a copy of nodes from the source tree
- `<xsl:value-of>`
  - add to the result tree the evaluation of an expression or the value of a source tree node
- `<xsl:for-each>`
  - reposition to a selection of source tree nodes or values using a supplied template

Instructions to push information from the source tree through the stylesheet:

- `<xsl:apply-templates>`
  - supply a selection of source tree nodes to push through template rules
- `<xsl:template>`
  - define a template rule

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

"If - Then" conditionality

Conditionally adding a template choosing from a single alternative:

```
<xsl:if test="expression-effective-boolean-value">
  template
</xsl:if>
```

- the effective Boolean value is the expression value converted implicitly to a boolean value
- test expression can be simple or complex
  - presence test with a node set expression
    - attempts to select from the source node tree according to the expression
      - determines success if at least one node from the tree would be selected or failure if no nodes from the tree would be selected
      - an empty node-set tests `false`; a non-empty node-set tests `true`
  - data type values can be tested as standalone values
    - the number values 0 and `NaN` test `false`, all other numbers test `true`
    - the empty string tests `false`, all other strings test `true`
    - a result tree fragment always tests `true` and can never test `false`
  - boolean operators can combine multiple criteria into a single testing alternative
- `true` test result adds template to the result tree
  - the XPath context for the template does not change
    - current node and current node list do not change
  - any instructions found therein are processed in the usual way
- `false` test result continues processing after instruction

An example of use with a node selection test expression:

```
<xsl:if test="caption"> <!--use long version if it has a caption-->
  <xsl:text>Figure title: </xsl:text>
  <xsl:value-of select="caption/long-title"/>
</xsl:if>
```

An example of use with a function test expression to create a comma-separated list:

```
<xsl:for-each select="//item">
  <xsl:value-of select="."/> <!--all in comma-separated list-->
  <xsl:if test="position()!=last()">, </xsl:if>
</xsl:for-each>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

"If - Else If - Else" conditionality

Conditionally adding a template choosing from multiple alternatives:

```
<xsl:choose>
  <xsl:when test="expression-effective-boolean-value">
    template
  </xsl:when>
  ...
  <xsl:when test="expression-effective-boolean-value">
    template
  </xsl:when>
  <xsl:otherwise>
    template
  </xsl:otherwise>
</xsl:choose>
```

- one or more <xsl:when> constructs
  - each with a boolean expression specified in a test= attribute
  - each tested in turn until first expression evaluates to true
    - the XPath context for the template does not change
      - current node and current node list do not change
    - any instructions found therein are processed in the usual way
  - the template may be empty
    - useful when necessary to not add anything in certain conditions and add something when all other conditions are tested
- at most one <xsl:otherwise> construct
  - used when all <xsl:when> constructs evaluate to false
  - following all <xsl:when> constructs
  - if absent when conditions require it to be used, nothing is added to the result

Note that at most one of the templates in the <xsl:choose> construct will be added to the result tree

- perhaps none
- never more than one

An example of use determining one's current location as the basis for presenting some text:

```
  <xsl:choose>
    <xsl:when test="self::frame">Frame </xsl:when>
    <xsl:when test="self::lesson">Lesson </xsl:when>
    <xsl:otherwise>Module </xsl:otherwise>
  </xsl:choose>
  <xsl:value-of select="title"/>
```

Of note:

- the first test true indicates which named node is on the self axis

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node type testing

Examples of using self:: axis to determine the type of the current node:

- the following tests can be read "if there are any nodes of the given node type or element node name along the self axis then add the template to the result tree"
- useful only for comment, processing instruction, text, and element nodes
  - the self:: axis cannot be used for attribute or namespace nodes because the primary node type along the axis is element node

```
 <xsl:choose>
   <xsl:when test="self::book:fig">     <!--an element in ns-->
     <xsl:text>book:fig: </xsl:text><xsl:value-of select="."/>
   </xsl:when>
   <xsl:when test="self::*">                  <!--an element-->
     <xsl:text>Element: </xsl:text><xsl:value-of select="."/>
   </xsl:when>
   <xsl:when test="self::text()">                    <!--text-->
     <xsl:text>Text: </xsl:text><xsl:value-of select="."/>
   </xsl:when>
   <xsl:when test="self::comment()">           <!--a comment-->
     <xsl:text>Comment: </xsl:text><xsl:value-of select="."/>
   </xsl:when>
   <xsl:when test="self::processing-instruction()">   <!--pi-->
     <xsl:text>PI: </xsl:text><xsl:value-of select="."/>
   </xsl:when>
 </xsl:choose>
```

Using self::book:fig axis is safer than name(.)='book:fig'

- the name(.) function returns the name with the namespace prefix as used in the XML source document
  - may be different than the namespace prefix used in the stylesheet
  - especially important when the XML source document is using the default namespace
  - it is very common for people to use name(.), but this technique is not namespace aware
- the self:: axis accommodates a namespace URI with an arbitrary prefix

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Node type testing (cont.)

P1 Examples of using the union operator to determine the current node:

- useful for root, attribute, and namespace nodes
- relies on the count of nodes being one when calculating the union of the current node with the desired node

```
 <xsl:choose>
   <xsl:when test="count(.|/)=1">                    <!--root-->
     <xsl:text>root </xsl:text>
   </xsl:when>
              <!--specific namespaced-unqualified attribute-->
   <xsl:when test="count(.|../@version)=count(../@version)">
     <xsl:text>version attribute </xsl:text>
   </xsl:when>
                  <!--specific namespace-qualified attribute-->
   <xsl:when test="count(.|../@book:ref)=count(../@book:ref)">
     <xsl:text>book:ref attribute </xsl:text>
   </xsl:when>
                       <!--any namespace-qualified attribute-->
   <xsl:when test="count(.|../@book:*)=count(../@book:*)">
     <xsl:text>book:* attribute </xsl:text>
   </xsl:when>
                                          <!--any attribute-->
   <xsl:when test="count(.|../@*)=count(../@*)">
     <xsl:text>attribute </xsl:text>
   </xsl:when>
                                     <!--specific namespace-->
   <xsl:when test="count(.|../namespace::xsl)=
                   count(../namespace::xsl)">
     <xsl:text>XSL namespace </xsl:text>
   </xsl:when>
                                          <!--any namespace-->
   <xsl:when test="count(.|../namespace::*)=
                   count(../namespace::*)">
     <xsl:text>namespace </xsl:text>
   </xsl:when>
 </xsl:choose>
```

The intuition of using `test="not(..)"` for the root node only works in XPath 1.0

- XPath 2.0 allows standalone attribute-node and namespace-node variables that do not have a parent, and thus would test true with "`not(..)`":

```
<xsl:variable name="ntattr" as="attribute()">
  <xsl:attribute name="book:ref" select="'test'"/>
</xsl:variable>
<xsl:variable name="ntns" as="node()">
  <xsl:namespace name="book" select="'urn:X-Crane:book'"/>
</xsl:variable>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node type testing (cont.)

T2 Node type testing is more intuitive in XPath 2.0:

- more node tests are available to use
- no node test for a namespace node, so same awkwardness is necessary as in XPath 1.0

Testing without consideration for the node's name:

```
<xsl:choose>
  <xsl:when test="self::element()">element</xsl:when>
  <xsl:when test="self::attribute()">attribute</xsl:when>
  <xsl:when test="self::comment()">comment</xsl:when>
  <xsl:when test="self::processing-instruction()">pi</xsl:when>
  <xsl:when test="self::text()">text</xsl:when>
  <xsl:when test="self::document-node()">doc</xsl:when>
  <xsl:when test="count( . | ../namespace::* )=
                  count( ../namespace::*)">ns</xsl:when>
</xsl:choose>
```

Testing with consideration for the node's name:

```
<xsl:choose>
  <xsl:when test="self::element(book:fig)">book:fig</xsl:when>
  <xsl:when test="self::attribute(book:ref)">@book:ref</xsl:when>
 <xsl:when test="self::processing-instruction(test)">p-test</xsl:when>
  <xsl:when test=". is ../namespace::book">ns-book</xsl:when>
</xsl:choose>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Example transformation requirement

To illustrate various aspects of the processing model, examine the different requirements presented by the following transformation of information from a source instance into a desired result in HTML.

Consider the following XML source instance `card.xml`:

```
01 <?xml version="1.0"?>
02 <?xml-stylesheet type="text/xsl" href="cardpush.xsl"?>
03 <card><name>G. Ken Holman</name>
04 <address>Box 266, Kars, Ontario CANADA K0A-2E0</address>
05 <email type="Current">gkholman@CraneSoftwrights.com</email>
06 <email type="Replaces">gkholman@CanadaMail.com</email>
07 </card>
```

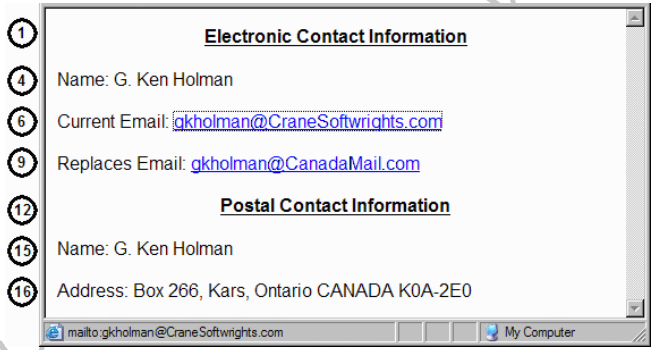
The source tree of nodes is as follows:

```
01 SHOWTREE - http://www.CraneSoftwrights.com/resources/
02 1  Proc. Inst. 'xml-stylesheet': {type="text/xsl" href="cardpush.xsl"}
03 2  Element 'card':
04 2.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
05 2.1  Element 'name' (card):
06 2.1.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
07 2.1.1  Text (card,name): {G. Ken Holman}
08 2.2  Text (card): {
09 }
10 2.3  Element 'address' (card):
11 2.3.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
12 2.3.1  Text (card,address): {Box 266, Kars, Ontario CANADA K0A-2E0}
13 2.4  Text (card): {
14 }
15 2.5  Element 'email' (card):
16 2.5.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
17 2.5.A  Attribute 'type': {Current}
18 2.5.1  Text (card,email): {gkholman@CraneSoftwrights.com}
19 2.6  Text (card): {
20 }
21 2.7  Element 'email' (card):
22 2.7.I  Namespace 'xml': {http://www.w3.org/XML/1998/namespace}
23 2.7.A  Attribute 'type': {Replaces}
24 2.7.1  Text (card,email): {gkholman@CanadaMail.com}
25 2.8  Text (card): {
26 }
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Example transformation requirement (cont.)

The objective is to render the following:



Of note:

- the numbers to the left correspond to the line numbers in the HTML file
- there is boilerplate text on the canvas that doesn't come from the source tree
- attribute values and element values are being displayed on the canvas
- the name is only in the instance once and is rendered twice
- each pass of the information duplicates some data and suppresses or selects other data
- the email addresses are hyperlinked (see status bar at bottom) thus requiring an element's content from the source to be included in both an attribute (with a prefix) and content of the result

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Example transformation requirement (cont.)

One HTML encoding is the following result instance:

```
<html>
<center>
<b><u>Electronic Contact Information</u></b>
</center>
<p>Name: G. Ken Holman</p>

<p>Current Email: <a href="mailto:gkholman@CraneSoftwrights.com"
>gkholman@CraneSoftwrights.com</a>
</p>
<p>Replaces Email: <a href="mailto:gkholman@CanadaMail.com"
>gkholman@CanadaMail.com</a>
</p>
<center>
<b><u>Postal Contact Information</u></b>
</center>
<p>Name: G. Ken Holman</p>
<p>Address: Box 266, Kars, Ontario CANADA K0A-2E0</p>
</html>
```

Of note:

- no line feed characters in the string before each anchor

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Approaches to transformation

Two approaches:

- to "pull" information from the source tree
  - adding the values into the result tree where desired
  - by obtaining the values of or repositioning to source tree nodes under transform control
- to "push" the source tree through the transform
  - the arrival of source tree nodes triggers repositioning events that are caught by template rules waiting in the stylesheet

Choosing which approach:

- depends on:
  - the nature of the source data
    - is the order known, unknown, or known but arbitrary?
    - is there mixed content that needs to be processed?
  - which file is to dictate the order of the result?
    - is the transform writer dictating the order through the transform?
    - does the transform have to relinquish control to the author of the XML document and have the source file dictate the order?
  - maintenance and flexibility questions
    - what language facilities are available for modularization and specialization?
- pull when the source data order is known
  - the kind and order of the nodes in the source tree are as expected by the transform writer
  - the transform writer dictates the creation of the result tree
    - the source nodes can be accessed on command
    - the data is pulled by obtaining values from each of the nodes at the required locations of the result
    - the count of source nodes at each location doesn't necessarily need to be known, only the location
      - it is possible to reposition to an unknown number of source nodes characterized only by their location

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Approaches to transformation (cont.)

Choosing which approach (cont.):

- push when the source data order is unknown or arbitrary
  - the source tree dictates the creation of the result tree
    - the kind and order of the nodes in the source tree are not expected by the transform writer
    - the writer chooses which portions of the source node tree are pushed through the transform
      - simply by name or by some other selection pattern for source tree nodes
    - the transform provides handlers for the events triggered by source tree nodes
      - the handlers are called template rules
        - the triggers are called match patterns
          - characterizes the source tree nodes being matched
        - groups of template rules can be named
          - engages a stylesheet-defined "mode" of transformation
          - possible to selectively engage a particular mode when pushing a given set of source tree nodes
      - the stylesheet may rely on built-in templates
        - not all source tree conditions have to be explicitly accommodated
  - supports transformation of mixed content
    - XML definition of character data interspersed with elements
  - promotes customization and adaptation of other stylesheets
    - fine-grained push-oriented templates can be exploited more easily than monolithic pull-oriented templates
  - supports source information from different document models
    - if it is necessary to accommodate different input hierarchies in one transform

Very often a mixture can best produce the result required:

- pushing nodes through node event handlers
  - preparing to accommodate the data in the order that it arrives
  - flexibly allows instances from different document models to be accommodated
- pulling nodes from within the node event handlers
  - pull information found in relative locations to those nodes that are being pushed
- handy loose rules of thumb:
  - push when handling branch nodes
  - pull when handling leaf nodes
  - not a cut-and-dried rule, but works well as an initial assumption

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes

Often necessary to copy source tree nodes to the result tree:

- recall the illustration on page 137
- in XSLT use a deep-copy instruction
  - `<xsl:copy-of select="`*`XSLT-expression`*`"/>`
  - `2` `<xsl:copy-of select="`*`XSLT-expression`*`" copy-namespaces="no"/>`
    - only copy necessary namespaces required to express the nodes being copied
    - unneeded namespaces for the element and its attributes are not copied
- copies all of the addressed nodes as nodes for the result tree (not values)
- copies all of the node's attached nodes and descendent nodes
  - recursively copies all child nodes and their children to the bottom of the tree

Deep-copying addressed nodes from source to result:

- e.g. two construction nodes followed by all `<p>` element children of all `<q>` elements in the source tree using the expression "`//q/p`"
- `<h1>Question</h1><p>Summary</p><xsl:copy-of select="//q/p"/>`
- if the copied nodes are elements, then the entire sub-tree below the element is copied

Convenient way to copy leaf nodes, e.g. all attributes:

- `<xsl:copy-of select="@*"/>`

Can accept any type of expression

- acts as `string()` or `<xsl:value-of/>` when the expression is not a node-set or result tree fragment, adding the value to the result as a string of text

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result text

Often necessary to add strings to the result tree:

- the "value of" instruction always returns a value of type string
- when in element content in the stylesheet use an instruction:
  - `<xsl:value-of select="`*`XSLT-expression`*`"/>`
  - 2 `<xsl:value-of select="`*`XSLT-expression`*`" separator="`*`AVT`*`"/>`
  - 2 `<xsl:value-of select="`*`expr-1,...,expr-2`*`" separator="`*`AVT`*`"/>`
  - 1 this instruction is always empty and never has a template
  - 2 `<xsl:value-of>`*`sequence-constructor`*`</xsl:value-of>`
    - converts a sequence of text nodes into a single text node
    - important where the cardinality of values must be one, not many
- when in attribute content in the stylesheet use an attribute value template (AVT):
  - `<`*`elem`* *`attr`*`="{`*`expr-1`*`}">`
  - 2 `<`*`elem`* *`attr`*`="{`*`expr-1`*`,...,`*`expr-n`*`}">`
  - typically used in the attributes of literal result elements
  - supported in some attributes of some XSLT instructions
    - most attributes of XSLT instructions are not attribute value templates
  - `<`*`elem`* *`attr`*`="http://{`*`expr-1`*`}/{`*`expr-2`*`}">`
    - multiple expressions in separate sets of brackets are allowed
    - brace brackets surround each expression
  - any XPath expression can be used
  - `<amt total="{($base + @fixed)*(1 + ancestor::order/@tax)}">`
    - an evaluated expression can be specified
    - the `total` attribute is calculated as the sum of the `base` variable and the current node's `fixed` attribute multiplied by the `order` ancestor's `tax` attribute
  - literal brace brackets "`{`" and "`}`" are specified as "`{{`" and "`}}`"

Important difference between nodes and strings in a sequence

- two consecutive strings (including strings created as values of element nodes or other data types) in a sequence are separated from each other with a space
- a node following a string, or two consecutive nodes, or a string following a node are not separated by a space

A node-set expression is used to access one or more values from a tree

- recall the string value of each kind of node (page 92)
- value calculated is the empty string if the set of nodes is empty
- 1 only first node in set (in document order) is used
- 2 all values in the set in sequence order (and document order within sequence member) are used
  - the separator string separates non-text-node members in the resulting string
  - the default separator is a single space
  - the default separator cannot be overridden in an attribute value template

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result text (cont.)

Consider the source fragment `<email type="Current">...</email>`:

- to use it in the following content of an element:
  `<p>Current Email: </p>`
  - using a literal result element in XSLT:

```
01  <p><xsl:value-of select="@type"/> Email: </p>
```

  or using an element instruction:

```
01  <xsl:element name="p">
02    <xsl:value-of select="@type"/>
03    <xsl:text> Email: </xsl:text>
04  </xsl:element>
```

  - in XSLT the node's value is equivalent to `data()`
    - in the absence of a schema and validation being turned on this is equivalent to `string()`

Recall the process diagram (page 137)

- element construction nodes in the operation tree have XML instance baggage
- element nodes copied from the operation tree to the result tree will carry the namespace nodes
- generated element nodes are created in isolation without any attached nodes
- will have all attached namespace nodes of ancestral namespace declarations that have not been explicitly excluded using `xsl:exclude-result-prefixes=`
  - e.g. those used for extension identification

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result text (cont.)

Chapter 4 - Processing model
Section 2 - Pull facilities

Often necessary to get information more than once:

- source XML:
  `<email type="Current">gkholman@CraneSoftwrights.com</email>`
- desired result:
  `<a href="mailto:gkholman@CraneSoftwrights.com"`
  `>gkholman@CraneSoftwrights.com</a>`
  - note how the element's content from the source has been used twice in the result
    - prefixed by "`mailto:`" in the attribute
    - in the element's content

Using a literal result element and attribute value template:

```
<a href="mailto:{.}">
  <xsl:value-of select="."/>
</a>
```

Using only instructions:

```
<xsl:element name="a">
  <xsl:attribute name="href">
    <xsl:text>mailto:</xsl:text>
    <xsl:value-of select="."/>
  </xsl:attribute>
  <xsl:value-of select="."/>
</xsl:element>
```

Note:

- when building an attribute, the processor knows the end result is a string, so an expression of multiple nodes adds the string value of each node
- the attribute to `<xsl:value-of>` is assumed to be an expression and therefore that attribute *cannot* be specified with an attribute value template
- the use of brace brackets for an attribute value template is only detected inside an attribute in the stylesheet
  - cannot be used in element content of the stylesheet even when defining attributes

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing result text

Chapter 4 - Processing model
Section 2 - Pull facilities

If appropriate to the output method, all sensitive characters added to the result tree are escaped when serialized

- '<' in text is escaped as '`<`' or '`&#60;`' or '`&#x3c;`' or '`&#x3C;`'
- '`&`' in text is escaped as '`&`' or '`&#38;`' or '`&#x26;`'
- '>' in text is escaped as '`>`' or '`&#62;`' or '`&#x3e;`' or '`&#x3E;`'
- using CDATA sections as in '`<![CDATA[<&>]]>`'
- '"' in an attribute may be escaped as '`&quot;`' or '`&#34;`' or '`&#x22;`'
- "'" in an attribute may be escaped as '`&apos;`' or '`&#39;`' or '`&#x27;`'

Requesting no escaping in XSLT:

- can request escaping be suppressed on any sequence of text added to result tree
- `<xsl:value-of select="`*`expression`*`"`
  `disable-output-escaping="yes"/>`
- the processor can choose to ignore the request
- escaping cannot be disabled when using attribute value templates
- use case is only for delivering non-XML markup (e.g. HTML) captured in XML
  - an XML instance containing HTML in `#PCDATA` necessarily escapes sensitive markup characters
    -
```
<pages>
  <page><![CDATA[
  <html>
    <body>
      <img src="picture.jpg">
      <hr>
    </body>
  </html>]]>
  </page>
  ...
</pages>
```
  - output escaping would escape sensitive markup resulting in no browser recognition of HTML, resulting in "`<img>`" instead of "`<img>`"
  - disabling the output escape serializes the content "as is" with sensitive markup characters ready to be recognized by downstream processes
- stylesheet writer takes the risk
  - disabling the escaping of characters can result in the output not being well-formed
  - not a typical requirement when transforming information for rendering purposes
  - not using this feature will result in all output being well-formed

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sample text generation

XSLT examples of obtaining source node tree values are as follows:

1 `<xsl:value-of select="name(.)"/>`
   - inject the expanded name '*`prefix`*:*`local`*' of the current node

2 `<xsl:value-of select="string(.)"/>`
   - inject the lexical value of the current source tree node

3 `<xsl:value-of select="data(.)"/>`
   - inject the schema-qualified value of the current source tree node

4 `<xsl:value-of select="."/>`
   - inject the schema-qualified value of the current source tree node

5 `<xsl:value-of select="$n//partNbr"/>`
   - 1️⃣ inject the value of the first of all descendent elements named `partNbr`
   - 2️⃣ inject the value of all descendent elements named `partNbr`, separated by a single space between each

6 `<xsl:value-of select="preceding-sibling::partNbr[1]"/>`
   - inject the value of the closest of all preceding sibling elements named `partNbr`
   - note the "`[1]`" is applied to the step, therefore it is in proximity order

7 `<xsl:value-of select="preceding-sibling::partNbr"/>`
   - 1️⃣ inject the value of the farthest of all preceding sibling elements named `partNbr`
      - note the "first" is applied to the path, therefore it is in document order
   - 2️⃣ inject the value of all preceding sibling elements named `partNbr`, in document order, separated by a single space between each

8
```
<xsl:value-of select="(//module[3]
                       /lesson[@type='exercise'][last()]
                       /frame/title)[1]"/>
```
   - inject the first title of a frame of the last lesson with an attribute node named "`type`" with the value "`exercise`", where that lesson is found in the third chapter
   - note the arbitrary use of white space between tokens of the XPath expression
   - important performance note
      - the use of "`//`" here is very wasteful in that the processor will look at *every* branch of the source node tree looking for `module` elements and will not just stop at the apex of the sub-tree beginning with `module`
      - while it is very intuitive to write "`//`", using this when it isn't needed could be the biggest source of poor performance
      - the time to use "`//`" is when it is necessary to visit every node in every branch of the source node tree (e.g. for index items)
      - some processors look for and optimize such behaviors with lookup tables

Recall the table of node values and names (page 92)
   - each node has a value and may have a name

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Repositioning using "pull"

One can reposition to each of a set of selected source tree nodes:

```
01  <xsl:for-each select="node-set-expression">
02    template
03  </xsl:for-each>
```

2️⃣ One can reposition to each of a sequence of arbitrary items (nodes or atomic values):

```
01  <xsl:for-each select="sequence-expression">
02    template
03  </xsl:for-each>
```

   - instruction element encapsulates a template to add for each new position
   - instruction attribute specifies which nodes or values make up the selected node set
   - the stylesheet dictates the order and content of the result tree

The context item focus "`.`" is the new position for the XSLT template

   - the `<xsl:for-each>` instruction's attribute is evaluated with the current node in the current context list of the template in which it is found
      - in the first example above "`@type`" is evaluated relative to the "`email`" element node, not to the parent node that was the current node when the repositioning was started
      - instructions *within* the template are evaluated using each of the nodes selected
         - the set of items selected becomes the current context list
            - the number of members is exposed in the `last()` function
         - each member of the set takes a turn as the current node
            - the position of each member is exposed in the `position()` function
         - any nodes are in document order or sequence order unless the first instructions of the template are sort instructions
   - when "`.`" is not a node, there is no definition of "`/`"
   - the evaluation context is restored at the end of `<xsl:for-each>` to that context in place when the instruction was executed

All items are visited without exception
   - to visit only a subset of items, only that subset must be specified in the expression
   - one often uses a predicate in the expression to specify which ones are included

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Repositioning using "pull" (cont.)

An example with nodes:

```
<xsl:for-each select="/card/email">
  <p><xsl:value-of select="position()"/> of
     <xsl:value-of select="last()"/>:
     <xsl:value-of select="@type"/> Email:
     <a><xsl:attribute name="href">
          <xsl:text>mailto:</xsl:text>
          <xsl:value-of select="."/>
        </xsl:attribute>
        <xsl:value-of select="."/>
     </a></p>
</xsl:for-each>
```

2 An example with atomic values:

```
<xsl:for-each select="'abc.com','def.net','ghi.org'">
  <a href="http://www.{.}">
     <xsl:text>http://www.</xsl:text>
     <xsl:value-of select="."/>
  </a>
  <xsl:if test="position()!=last()">, </xsl:if>
</xsl:for-each>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Card sample pull transforms

Recall the transformation objective (page 145) using the given data (page 144)

The file `cardpull.xsl` illustrates the pull approach using a simplified stylesheet with only a result tree template:

```
<?xml version="1.0"?><!--cardpull.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
  <center><b><u>Electronic Contact Information</u></b></center>
  <p>Name: <xsl:value-of select="/card/name"/></p>
  <xsl:for-each select="/card/email">
    <p>
      <xsl:value-of select="@type"/>
      <xsl:text> Email: </xsl:text>
      <a href="mailto:{.}"><xsl:value-of select="."/></a>
    </p>
  </xsl:for-each>
  <center><b><u>Postal Contact Information</u></b></center>
  <p>Name: <xsl:value-of select="/card/name"/></p>
  <p>Address: <xsl:value-of select="/card/address"/></p>
</html>
```

Of note:

- the "`/card/*`" addresses could equally as well have been "`card/*`" addresses because the current node for the transformation is the root node
- no use of `<xsl:apply-templates/>` because there are only built-in template rules and no user template rules in a simplified stylesheet

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Repositioning using "push"

Classical event-processing scheme in XSLT

Generate zero to many events:

```
01  <xsl:apply-templates select="node-set-expression">
```

- selects the nodes that dictate the construction of the next portion of the result tree
- the set of nodes selected becomes the current node list for the processing of each node
  - each member of the set takes a turn as the current node
- the nodes are in document order unless the first instructions of the template are sort instructions
- the `select=` attribute is optional
  - assumes `select="node()"` when absent
    - the current node list becomes all child nodes of the current node
  - attribute and namespace nodes of the source node tree *are not implicitly visited*
    - must explicitly be visited under stylesheet control by selecting the nodes desired
- the data dictates the order and content of the result tree
- the evaluation context is restored at the end of `<xsl:apply-templates>` to that context in place when the instruction was executed
- typically used only inside branch (root and element) templates, not inside leaf templates
  - leaf nodes have no child nodes to be processed
  - not an error to ask non-existent child nodes to be processed
  - might be used to from a leaf node to process absolutely-addressed nodes, nodes relative to the parent node, or the current leaf node in another collection of template rules
- T2 nodes from a temporary tree variable can be selected
  - unlike the restriction for XSLT 1.0 of not selecting result tree fragment nodes

All nodes are pushed at the stylesheet, without exception

- to push only a subset of nodes, only that subset must be specified in the expression
- one often uses a predicate in the expression to specify which ones are included

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Repositioning using "push" (cont.)

Handle events:

```
01  <xsl:template match="node-set-pattern">
02    template
03  </xsl:template>
```

- the template constructing the next part of the result tree when given nodes are visited
  - the `match=` pattern does not need to be equivalent to the `select=` pattern that was used in `<xsl:apply-templates>`
  - nodes and their contexts as described in the pattern are evaluated in the context of the source tree
- the template rules can be grouped
  - collections of template rules called "modes"
    - described later in this chapter
- built-in template rules can be assumed
  - every node matched even if no match in the stylesheet
    - described later in this chapter
- template conflict resolution rules apply for ambiguous template matching
  - cannot have more than one event handler that is a candidate for being triggered for visiting a given node (described later in this chapter)
  - the nature of the pattern and additional attributes in the stylesheet control matching nuances
- the template rules can be prioritized
  - the shape of the pattern governs implicit priority
    - described later in this chapter
- T2 the template rule's template result can be constrained
  - the nodes created as a result of processing the template can be checked for correctness
    - described in Chapter 6 Transform and data management (page 209)
- the template rule can be named
  - invoked on demand by the stylesheet as well as required by the nodes selected from the source node tree
    - described in Chapter 6 Transform and data management (page 209)
- the template rule's template can be parameterized
  - variables whose bound values can be passed by the process invoking the template
    - described in Chapter 6 Transform and data management (page 209)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Repositioning using "push" (cont.)

For example, in the objective of this sample the processing of the root node can ask the XSLT processor to look for template rules for all child element nodes named "card", thus the document element must be named "card" (reduces the chance of stylesheet abuse):

```
01 <xsl:template match="/"> <!--process root node-->
02   <xsl:apply-templates select="card"/> <!--assume doc. element name-->
03 </xsl:template>
```

The XSLT processor finds the template rule for a document element named card, adds to the result tree and then processes child nodes:

```
01 <xsl:template match="card">
02   <center><b><u>Electronic Contact Information</u></b></center>
03   <xsl:apply-templates/>
04 </xsl:template>
```

In turn, having found an email child element node, the processor will find the template rule for that element:

```
01 <xsl:template match="email">    <!--generate a mailto:-->
02   <p><xsl:value-of select="@type"/>
03     <xsl:text> Email: </xsl:text>
04     <a href="mailto:{.}">
05       <xsl:apply-templates/>
06     </a>
07   </p>
08 </xsl:template>
```

In this recursive-like process:

- the <xsl:template> constructs set up the event handlers to handle the set of nodes selected in <xsl:apply-templates>
  - either explicitly a node set using a select= expression or implicitly just the child nodes
- the context continually is changed and restored as the node tree is traversed
- the data dictates the order in which templates are processed
- each template is added to the result tree thus creating the result tree parse order

The order of template rules is *not* important (except in error recovery)

- all template rules are at the "top" of the stylesheet tree, so have equal weight
- order comes into play only for error recovery in ambiguous template matching

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Repositioning using "push" (cont.)

Recall that every XML document can be input to any XSLT stylesheet

- e.g. an instance with <invoice> can be passed to the example stylesheet
- without an explicit push in a template for the root node, the built-in template rules will automatically engage the nested template processing

Pushing the document element from the root note protects the stylesheet from abuse

- only instances with <card> produce non-empty output

Resulting file not acceptable as a document entity for use downstream

- a document entity must have a document element to be well-formed
- an empty file is still acceptable as an external parsed general entity

Namespaces further protect a stylesheet from abuse

- using only <card> without namespaces could be ambiguous
- consider a greeting card instances as follows:

```
01 <card xmlns="http://www.mycompany.org/ns/greeting">
```

- and a business card instances as follows:

```
01 <card xmlns="http://www.mycompany.org/ns/business">
```

- these can be distinguished in a business card stylesheet as follows:

```
01 xmlns:bc="http://www.mycompany.org/ns/business"
02 ...
03 <xsl:template match="/">
04   <xsl:apply-templates select="bc:card"/>
05 ...
06 <xsl:template match="bc:card">
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Empty templates

Chapter 4 - Processing model
Section 3 - Push facilities

Sometimes necessary to add nothing to the result tree:

- all events must be handled even if not wanted
  - built-in template rules may add information to the result tree when not desired
  - a stylesheet template rule overrides a built-in template rule
- no different in principle than a non-empty template
  - the instruction itself is empty (i.e. has no children), so the empty template is added to the result

```
01                                  <!--suppress the address-->
02 <xsl:template match="address"></xsl:template>
```

According to XML rules for empty elements, the above markup is no different than the following (though the preceding may be more easily comprehended by a reader of the stylesheet):

```
01 <xsl:template match="address"/> <!--suppress the address-->
```

According to XSLT rules for ignored comments and white-space-only text nodes, the above markup is no different than the following (though they are distinctly different in XML terms):

```
01 <xsl:template match="address">  <!--suppress the address-->
02 </xsl:template>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Modes

Chapter 4 - Processing model
Section 3 - Push facilities

Collections of template rules can be made:

- mode="*mode-qname*"
  - in <xsl:apply-templates/> to indicate which collection to use
  - in <xsl:template> to indicate which collection of membership
- when mode= is not specified, the unnamed collection "#default" is used
  - 1.1 the "#default" mode cannot be specified explicitly
- 2.0 the initial mode for the processing of the root node can be specified
  - when not specified, the "#default" mode is used
- 2.0 <xsl:apply-templates/> can use mode="#current"
  - the mode in play when the last match occurred is used
- 2.0 <xsl:template> can use mode="#all" or mode="*mode-list*"
  - the template matches based on match criteria regardless of the mode in play

For example, consider in the objective to process one set of templates for the electronic information and a second set of templates for the postal information where the same match patterns are needed in both processes:

```
01 <xsl:template match="/">        <!--process entire instance twice-->
02   <xsl:apply-templates select="card"/>
03   <xsl:apply-templates select="card" mode="postal"/>
04 </xsl:template>
05                                 <!--electronic info-->
06 <xsl:template match="card">
07   <!--template here-->
08 </xsl:template>
09                                 <!--postal info-->
10 <xsl:template match="card" mode="postal">
11   <!--template here-->
12 </xsl:template>
```

The root rule above processes the document element twice, once in each of two different modes

- every execution of <xsl:apply-templates> will only use those <xsl:template> rules with the matching mode attribute

Without modes each template would have complex context calculations

- by using separate collections each collection can be treated as a mini-stylesheet
- modes can be changed at any time
- there can be any number of modes each reflecting different requirements of the stylesheet without having to accommodate other requirements

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Built-in template rules

XSLT specifies built-in template rules that are assumed to be a part of all stylesheets

- when the processor can't match a stylesheet template rule to a given node, a built-in template rule is used
- the built-in templates are documented below using XSLT 2.0 concepts of "`#all`" and "`#current`" but the documentation applies equivalently (just not as elegantly) to built-in templates specified in XSLT 1.0

When there is no explicit template rule for the root node or an element node, the following built-in rule will be assumed by the XSLT processor with an implicit `mode=` attribute of the current mode:

- note how attributes are not pushed by built-in templates, they must be explicitly pushed or processed by the stylesheet

```
01 <xsl:template mode="#all" match="*|/">
02   <xsl:apply-templates mode="#current"/>
03 </xsl:template>
```

- 1 in XSLT 1.0 any passed parameters are not passed along
- 2 in XSLT 2.0 all passed parameters are passed along

When there is no explicit template rule for a text or attribute node, the following built-in rule will be assumed by the XSLT processor to add the leaf node's value to the result tree:

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

When there is no explicit rule for a comment or processing-instruction node, the following empty built-in rule will be assumed by the XSLT processor and, when triggered, it specifies that nothing is to be added to the result tree:

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

All namespace nodes are also ignored (as there is no pattern with which to recognize them in a match attribute). Namespace nodes are added to the result tree piggybacked on element nodes being copied to the result tree and as a result of instructions synthesizing result tree nodes.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Template rule conflict resolution

Template rule selection must be unambiguous:

- separate collections of rules are distinguished by the mode used at select time
- more than one matching rule in a given collection of rules is an error
  - the XSLT process must only match a single template rule for each event
  - it is the stylesheet writer's responsibility to not confuse the processor
    - must only supply a single template rule for every event or rely on the built-in templates for an event not explicitly handled
- multiple hierarchical relationships can often be ambiguous
  - template rule matching patterns are often written with hierarchies specifying the context of nodes
  - when one hierarchy is a strict subset of the other hierarchy then both hierarchies are candidates for a node that is found in the subset

Important portability issue

- the XSLT processor is *not* required to report a template conflict error condition
  - if not reported, it is required to use the *last* template rule found in the stylesheet that matches the source tree node
  - a stylesheet apparently successfully running using one XSLT processor not reporting template conflict errors *will not run* in another XSLT processor that chooses to report such errors

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Template rule conflict resolution (cont.)

Every template rule has a priority:

- used by the XSLT processor to avoid conflict within a given collection of template rules
  - helpful when specifying multiple template rules with subsets of matching conditions
- highest priority wins
  - the processor chooses the matching template rule within the given collection that has the highest explicit or implicit numeric priority value
- explicit priority specified by user
  - a floating point number value
  - `<xsl:template match="`*`match-pattern`*`" priority="`*`numeric-value`*`">`
- implicit priority assumed based on pattern specificity when `priority=` is not specified:
  - -.5 for patterns comprised of *only* a wildcard or node kind
    - wildcards: "`*`" or "`@*`"
    - 2 wildcards: `element()`, `element(*)`, `attribute()`, `attribute(*)`
    - node types: `node()`, `comment()`, `processing-instruction()` or `text()`
  - -.25 for patterns comprised of a namespace prefix and a wildcard
    - "*`prefix`*`:*`"
    - 2 `*:`*`prefix`*
  - 0 for patterns comprised of *only* a node's name or *only* a data type
    - un-prefixed or `child::` axis for an element
    - prefixed with "`@`" or `attribute::` axis for an attribute
    - with or without a namespace prefix
    - a literal in `processing-instruction(`*`PI-target`*`)`
    - 2 the patterns `element(`*`name`*`)`, `element(*,`*`type`*`)`, `attribute(`*`name`*`)` or `attribute(*,`*`type`*`)`
  - 2 +.25 for patterns comprised of *both* a name and a data type
    - the patterns `element(`*`name`*`,`*`type`*`)`, `element(`*`name`*`,`*`type`*`)`, `attribute(`*`name`*`,`*`type`*`)` or `attribute(`*`name`*`,`*`type`*`)`
  - +.5 for all other patterns
  - nuance between XSLT 1.0 and XSLT 2.0 matching a document node "`/`"
    - 1 value of +.5 in XSLT 1
    - 2 value of -.5 in XSLT 2

A union match pattern for a template rule is handled severally

- the processor will regard each of the patterns separately as if they were present individually in copies of the template rule

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Template rule conflict resolution (cont.)

Consider an example of a typographical vocabulary

- the element `figure` specifies a figure reference with a `security` attribute
- the element `para` specifies a paragraph of content
- the elements are in the vocabulary identified by a book URI

```
<book:para security="classified">
  Components: <book:figure security="top" image="missile.gif"/>
</book:para>
```

The rendering of figures differs based on context

- when found outside a paragraph
  - perhaps it is rendered centered on the line, with a hotlink to a description
- when found inside a paragraph
  - perhaps it is rendered inline to the paragraph, with a hotlink to a description
- when it has a security attribute regardless of context
  - perhaps it is to be elided entirely if considered top secret
- each of the above contexts requires a different template rule, but all must be active to accommodate all of the different situations
  - can only be distinguished unambiguously by using priority

Without priority, all of the following rules would conflict for the processing of a figure when that figure is in a paragraph and it has the security attribute value of "`top`":

```
<xsl:template match="b:figure[@security='top']" priority="2"/>

<xsl:template match="b:para/b:figure"><!--implied priority=".5"-->

<xsl:template match="b:figure"><!--implied priority="0"-->

<xsl:template match="b:*"><!--implied priority="-.25"-->

<xsl:template match="*"><!--implied priority="-.5"-->
```

Note that without explicit priority:

- the first two rules above would both have the implicit value of .5 for priority
- a processor not reporting the conflict would always process the second of the two rules for figures found in paragraphs

Rule of thumb:

- the more specific or important the matching pattern relative to other candidate matching patterns, the higher the specified priority must be

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

General approach to writing templates

As a general rule of thumb when deciding how to write a template rule:

- when matching different nodes of different names: no special approach
  - modes, context and priority need not come into play when template rules have no overlapping match conditions
- when matching the same node for different results: use different modes
  - without modes there would be a template conflict (which may not get reported)
  - need to distinguish the different results in templates of different collections that get matched using different modes
- when matching different nodes with the same name: use context and possibly priority
  - the use of context distinguishes nodes from each other by their respective ancestors or by respective predicates
  - the use of priority distinguishes templates when the implicit template priority is not unique

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Card sample push transforms

Recall the transformation objective (page 145) using the given data (page 144)

The file `cardpush.xsl` illustrates the push approach using an traditional stylesheet with a number of template rules and named modes (collections of template rules):

```
01 <?xml version="1.0"?><!--cardpush.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03 <xsl:stylesheet version="1.0"
04      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
05
06 <xsl:template match="/">     <!--process tree twice-->
07  <html><xsl:apply-templates select="card"/>
08        <xsl:apply-templates select="card" mode="postal"/></html>
09 </xsl:template>
10                                  <!--electronic info-->
11 <xsl:template match="card">
12  <center><b><u>Electronic Contact Information</u></b></center>
13  <xsl:apply-templates/></xsl:template>
14
15 <xsl:template match="name">
16  <p>Name: <xsl:apply-templates/></p></xsl:template>
17
18 <xsl:template match="address"/> <!--suppress address-->
19
20 <xsl:template match="email">    <!--generate a mailto:-->
21   <p><xsl:value-of select="@type"/>
22      <xsl:text> Email: </xsl:text>
23      <a href="mailto:{.}"><xsl:apply-templates/></a></p>
24 </xsl:template>
25                                  <!--postal info-->
26 <xsl:template match="card" mode="postal">
27   <center><b><u>Postal Contact Information</u></b></center>
28   <xsl:apply-templates select="name"/>
29   <xsl:apply-templates select="address" mode="postal"/>
30 </xsl:template>
31
32 <xsl:template match="address" mode="postal">
33   <p>Address: <xsl:apply-templates/></p></xsl:template>
34
35 </xsl:stylesheet>
```

Of note:

- every `<xsl:apply-templates/>` must specify that mode to be used for the template matches
  - the template for `name` is used by the `<xsl:apply-templates>` instructions on line 13 and line 28
- the document element is visited twice in the first template (lines 7 and 8)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Processing model summary

The XSLT processor must produce the same effect as the following:

- build the operation node tree from the stylesheet file
  - operation tree contains instruction elements and literal result elements
- build the primary source tree from the source resource
  - to act on with directives found in the transform regarding white-space handling
- implicitly execute `<xsl:apply-templates select="/"/>`
  - [1][2] can alternatively specify the starting mode or named template rule
  - instantiate the template for the root node as the start of construction of the result tree
  - note that in an implicitly declared stylesheet, the one result template found in the stylesheet is assumed to be the template for a template rule for the root node, thus satisfying the above instruction
- for each `<xsl:apply-templates>` and `<xsl:for-each>` instruction:
  - `select=` defines the current context list
  - the current context list may be filtered
    - by using a predicate in the `select=` expression
  - the current context list may be sorted
    - described in Chapter 9 Sorting and grouping (page 401)
  - for all of the items in the current context list
    - visit each item in turn as the current item
    - determine the template to be used to construct "the next part" of the result node tree
      - if using `<xsl:apply-templates>`:
        - match the template rule of highest priority in the `mode=` collection
        - pass all parameterized variable binding values (described in Chapter 6 Transform and data management (page 209))
      - if using `<xsl:for-each>`:
        - use the template of the instruction itself
    - add literal result elements and results of operation execution to the result tree
      - replace operation in situ with the template result of instruction execution
- read other source files or data sources into node trees as instructed

Note that the processor is not required to be implemented explicitly in any particular way:

- a given implementation can produce the identical result following any algorithm
- the recommendations describe the desired end effect, not the method of implementation

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Parallelism

An XSLT processor implementation has the latitude to process the input in any fashion, including in a parallel fashion, as long as the result is the same as when processed serially as described in the Recommendation:

- the side-effect free nature of the XSLT expression language allows an implementation to simultaneously process multiple nodes of the tree without impacting on expression evaluation
  - by design, the processing of any one template rule on a given node of the source tree cannot have any impact on the processing of any other template rule or the same template rule on any other node of the source tree
- the end result of the parallel processes must be assembled *as if* the source tree nodes were processed sequentially

For example, consider a book with a number of chapters:

- each chapter node can be processed in parallel on different computers or processors
- the completed result sub-trees for each chapter are assembled in the order of the corresponding source tree chapter nodes
- the end result is achieved in a time related to the longest processing of one chapter, not on the sum of the processing of all chapters

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Suggested stylesheet development approach

Chapter 4 - Processing model
Section 4 - Summary and examples

Always remember it is the stylesheet's responsibility (thus the stylesheet writer's responsibility) to create the result tree in result tree document order:

- what is the document element of the result?
- what is the first content that belongs in the result?
- what comes next, and next, and next, to the end of the result?
- for each piece of content that belongs in the result
  - is it found in the source tree?
    - if so, then that part of the source tree must be the next part that is accessed and processed
    - if not, then the content must be put directly in the stylesheet for adding to the result
  - ensure all attribute nodes for a result element are added before its content

An iterative approach to stylesheet development:

- begins by developing templates with instructions that handle high level constructs in the source node
- progresses by developing templates for successively lower level constructs within each high level construct
- shows small results quickly with simple stylesheets before growing into complex stylesheets
- will allow you to build on previous successes and recognize pitfalls early

Note that even if some high level constructs will be invisible in the final stylesheet, revealing their presence in the output at early stages of development helps with diagnosing problems in the templates.

The following is a helpful technique when working with an XSLT processor that does not report template conflict errors:

- in the physical order of templates in the stylesheet file, order templates with more specificity before templates with less specificity
- as new templates are added, neglecting to attach priority will produce no change to the results as the processor will fall back to the latter templates (thus indicating the newly added template isn't being processed and requires priority)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 5 - Transformation environment

- Introduction - The transformation environment
- Section 1 - Transform properties
- Section 2 - In-scope schema specifications
- Section 3 - Serialization
- Section 4 - Communicating with the outside environment

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

The transformation environment

Chapter 5 - Transformation environment

Different ways are available to communicate to and from a processor

- some aspects of transformation are under transform control
- others cannot be manipulated under transform control

XPath 2 functions for diagnostics

- P2 `error()`
  - signaling a premature end of process
- P2 `trace()`
  - diagnostic reporting of function values

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The transformation environment (cont.)

Chapter 5 - Transformation environment

The XSLT instructions covered in this chapter are as follows.

Wrapping the content of a stylesheet:

- `<xsl:stylesheet>`
  - encapsulate a stylesheet specification
- `<xsl:transform>`
  - encapsulate a stylesheet specification

Schemas and serialization:

- `<xsl:namespace-alias>`
  - specify a result tree namespace translation
- `<xsl:output>`
  - specify the desired serialization of the result tree
- T2 `<xsl:character-map>`
  - specify a translation of characters during serialization
- T2 `<xsl:output-character>`
  - specify a translation of single character during serialization
- T2 `<xsl:import-schema>`
  - gain the awareness of user-defined data types
- T2 `<xsl:result-document>`
  - create more than a single result tree

Communicating with the operator:

- `<xsl:message>`
  - report a stylesheet condition to the operator
- `<xsl:param>`
  - supply a parameterized value from the operator

The functions covered in this chapter are as follows.

Environment functions:

- `system-property()`
  - accessing system-defined property strings
- T2 `type-available()`
  - accessing system-defined property strings

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element

An XSLT stylesheet is an XML document and must be XML-well-formed

- makes it easy for an XSLT stylesheet to be the input and/or the output of other XML tools, including XSLT
- the document element must declare "`http://www.w3.org/1999/XSL/Transform`" (case sensitive)
  - the namespace prefix is arbitrary and need not be "`xsl`"
    - historically, XSLT was first specified as chapter 2 of the XSL specification, thus having the "`xsl`" prefix gaining popularity amongst users
- 1 XML default namespace not used in XPath element addresses
  - all element names in non-null namespaces must use a namespace prefix
- 2 XML default namespace may be declared for XPath element addresses
  - by default all un-prefixed element names in XPath are for names in no namespace
  - see page 180 for `xpath-default-namespace=` to override this behavior

A simplified stylesheet has a non-XSLT document element (e.g. page 49)

- the entire document represents the template of the template rule of the root node
- must declare the version of XSLT used by the stylesheet:
  - *`prefix`*`:version="`*`number-value`*`"`

Identical and interchangeable choices for the XSLT document element (e.g. page 50)

- the prefix can be omitted in order to use the default namespace
  - default namespace not recommended because access to XSLT attributes in literal result elements needs a prefix
- `<`*`prefix`*`:stylesheet>`
- `<`*`prefix`*`:transform>`
- must declare the version of XSLT used by the stylesheet
  - `version="`*`number-value`*`"`
  - the prefix is not needed because the attribute is in an instruction
- non-XSLT children of document element allowed for user data
  - document element child elements must be in a non-XSLT namespace
  - can be accessed using the `doc()`or `document()` function easily

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element (cont.)

Can also be used as a container element inside an XML instance

- for a stylesheet embedded in another context
  - may use `id="`*`unique-identifier`*`"`
    - identifies the stylesheet when there are multiple ones from which to choose
    - the use of this XML `ID` attribute is outside the scope of the recommendation
    - could be used as a fragment identifier by the stylesheet association processing instruction or by other techniques to identify a given stylesheet among many

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

The stylesheet document/container element (cont.)

Non-XSLT namespace declarations are used for literal result elements, vendor extensions, named constructs, data types, schema types and modes

- literal result elements are examples of portions of the result tree
  - e.g. `xmlns="http://www.w3.org/1999/xhtml"`
- declarations to recognize constructs from the source node tree
  - e.g. `xmlns:in="urn:oasis:...:xsd:Invoice-2"`
- vendors publish namespaces (not prefixes) to identify functionality
  - e.g. `xmlns:sx="http://icl.com/saxon"` (for Saxon 6)
  - e.g. `xmlns:sx="http://saxon.sf.net/"` (for Saxon 9)
  - e.g. `xmlns:cts="http://marklogic.com/cts"` (for MarkLogic)
- named constructs (e.g. variables, templates, keys, etc.), modes and top-level user elements can be distinguished using namespaces
  - e.g. `xmlns:xs="http://www.CraneSoftwrights.com/ns/xslstyle"`
- data types and user schemas
  - e.g. `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`

Controls available on container element or any literal result element:

- scope of influence is all descendent elements in stylesheet
  - these attributes as attributes of instructions do not need to be in the XSLT namespace
    - e.g. `<xsl:instruction exclude-result-prefixes="sx">`
  - these attributes as attributes of literal result elements need to be in the XSLT namespace to be distinguished from the attributes targeted for the result tree
    - e.g. `<l-r-element xsl:exclude-result-prefixes="sx">`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element (cont.)

Controls available (cont.):

- `exclude-result-prefixes="white-space-separated-prefixes"`
  - indicate which stylesheet namespace prefixes are not expected in the result, thus are not to be included in the stylesheet tree on literal result elements
    - a list of white-space-separated namespace prefixes specifying prefixes that are to be explicitly excluded from the stylesheet tree (using `#default` as the name to reference the default namespace, which is sometimes unofficially called the null namespace)
    - 2 `exclude-result-prefixes="#all"` is allowed
    - e.g. `exclude-result-prefixes="sx in cac cbc ext sig sac sbc"`
  - user-specified prefixes and associated namespace declarations are often used in XSLT stylesheets (but not desired in the result) for various purposes such as:
    - top-level documentation
    - embedded structured data
    - named XSLT constructs
  - recall that copying an element node from the stylesheet to the result will copy all attached namespace nodes thus stylesheet namespace declarations can easily end up in the result tree
  - this exclusion declaration tells the XSLT processor to not include the specified namespace nodes on descendent nodes of the stylesheet tree
  - this exclusion declaration has no effect on namespace nodes of the source tree
- `extension-element-prefixes="white-space-separated-prefixes"`
  - indicate which stylesheet namespace prefixes are instruction prefixes
    - a list of white-space-separated namespace prefixes specifying prefixes that are extension namespaces to be recognized by the XSLT processor (using `#default` as the name to reference the default namespace)
  - recall that everything that is not an instruction is considered to be a literal result element
  - elements prefixed with the namespace prefix associated with the XSLT URI are interpreted as instructions
  - this declaration tells the XSLT processor what other prefixes are to be interpreted as instructions because they are extension elements required by the stylesheet
  - the processor need not implement the extension elements (detailed in Chapter 6 Transform and data management (page 209))
  - not needed if only using extension functions and not extension elements

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element (cont.)

Controls available in XSLT 2.0

- `xpath-default-namespace="`*`uri-string`*`"`
  - supplies a URI to use for un-prefixed element names used in XPath addresses
  - recall the distinction between XML and XPath default namespaces (page 122)
  - allowed anywhere in the tree, with influence through descendent constructs
- `default-collation="`*`uri-list`*`"`
  - supplies a list of URI strings from which an implementation must recognize one to use for XPath expressions
    - specifies the rules for comparing strings
      - e.g. a collation where "ß" (German sharp-s) is ordered between "`s`" and "`t`"
      - see page 289 for details
  - this does not affect XSLT collations used by `<xsl:sort>`
  - to ensure at least one of the strings is recognized, one could end the URI list with the Unicode Codepoint Collation URI
    - `http://www.w3.org/2005/xpath-functions/collation/codepoint`
  - allowed anywhere in the tree, with influence through descendent constructs
- `default-validation="`*`preserve-or-strip`*`"`
  - governs the default behavior when building the result tree from nodes of a source tree
  - this does not affect nodes that are constructed from scratch using XSLT instructions
  - using "`preserve`" will keep the data type of nodes when copied to the result tree
  - using "`strip`" will change the data type of an element to `xs:untyped` and that of an attribute to `xs:untypedAtomic`
  - this attribute does not invoke schema validation
    - to invoke schema validation on the entire result tree, wrap the result tree document element in an `<xsl:document>` instruction
  - allowed anywhere in the tree, with influence through descendent constructs

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element (cont.)

Controls available in XSLT 2.0 (cont.)

- `input-type-annotations="`*`preserve-or-strip-or-unspecified`*`"`
  - governs the use of type annotations attributed to the source tree nodes
  - a value of "`strip`" will remove all data type information from elements and attributes and turn off the translation of the string value to the typed value
    - elements become `xs:untyped`
    - attributes become `xs:untypedAtomic`
    - values of elements and attributes become `xs:untypedAtomic` strings
  - this does not impact other schema-based information such ID/IDREF information or defaulted values
  - default value if not used is "`unspecified`"
    - allows the attribute to be specified when you don't want to change what another fragment may wish to use
  - in one stylesheet no two stylesheet fragments can have contrasting values of "`strip`" and "`preserve`" in their document elements
  - only allowed in the document element, not elsewhere in the tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The stylesheet document/container element (cont.)

Child elements of the document or container element are "top-level" elements:

- if present, the following must occur before all other top-level elements
  - `xsl:import`
    - see Imported stylesheets (page 246)
- if present, the following (listed alphabetically) may occur in any order as top-level elements
  - `xsl:attribute-set`
    - see Constructing attribute nodes (page 362)
  - `xsl:character-map`
    - see Character maps (page 199)
  - `xsl:decimal-format`
    - see Decimal formatting in XSLT (page 295)
  - `xsl:function`
    - see User-defined functions (page 240)
  - `xsl:import-schema`
    - see Importing schema definitions (page 185)
  - `xsl:include`
    - see Included stylesheets (page 245)
  - `xsl:key`
    - see XSLT key node referencing (page 320)
  - `xsl:namespace-alias`
    - see Namespace protection (page 187)
  - `xsl:output`
    - see Serializing the result tree (page 191)
  - `xsl:preserve-space`
    - see White-space-only text nodes (page 88)
  - `xsl:strip-space`
    - see White-space-only text nodes (page 88)
  - `xsl:template`
    - see Repositioning using "push" (page 159)
  - the following are not only used as top-level elements, while all others listed above are only used as top-level elements
    - `xsl:param`
      - see Variable and parameter binding (page 225)
    - `xsl:variable`
      - see Variable and parameter binding (page 224)
- top-level elements in a non-XSLT namespace are ignored
  - useful for stylesheet data for access during the transformation
  - useful for richly-structured supplemental documentation (e.g. see page 525)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Documenting stylesheets

Because an XSL stylesheet is an XML document:

- XML comments can be used to provide documentation about the stylesheet
- all XML comments and processing instructions found in an XSL stylesheet are ignored
  - note that some XML editing tools may leave processing instructions in files for remembering locations such as the last cursor position

Adding richly marked up documentation to a stylesheet:

- allows the stylesheet to be run through a documenting stylesheet to extract the documentation in any fashion desired
- is accomplished by including non-XSLT constructs as top-level elements (children of the stylesheet document element) provided that the default namespace is not used as the namespace for such constructs, as in the following example:

```
01 <?xml version="1.0"?><!--hellodoc.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03
04 <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05                version="1.0" exclude-result-prefixes="mydoc"
06                xmlns:mydoc="http://www.mycompany.com/mydoc">
07
08 <xsl:output method="html"/>
09
10 <mydoc:para>
11 The following construct is the root template.
12 </mydoc:para>
13
14 <xsl:template match="/">                          <!--root rule-->
15     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
16     <?test a processing instruction here?>
17 </xsl:template>
18
19 </xsl:transform>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Documenting stylesheets (cont.)

Note the use of `exclude-result-prefixes=` in the document element above to tell the XSLT processor to not emit a namespace declaration for the prefix of the documentation namespace:

- if the stylesheet writer knows that namespace will never be needed in the result
- because the XSLT processor doesn't know when creating the document element node of the result tree whether the namespace will ever be needed in the instance, so by default the declaration is emitted

See the Crane Softwrights Ltd. web site free resources section for an integrated XSLT stylesheet documentation environment named XSLStyle™:

- a stylesheet for stylesheets to render embedded DocBook (or any other vocabulary) as top-level constructs in XSLT

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Importing schema definitions

User-defined schema types are specified by external or internal schema expressions

- there are no pre-defined schema types
- an error is triggered if a non-schema-aware processor attempts to perform validation

T2 Declare the schemas at the top level of the stylesheet

- `<xsl:import-schema>`
- `namespace=`
  - specifies the namespace of the schema expression
  - this must match any present declared namespace within the schema
- `schema-location=`
  - makes reference to an external schema expression
- alternative W3C Schema expression content defines a synthetic schema:
  - exclusive with `schema-location=`

An example adapted from the XSLT 2.0 specification

```
01 <xsl:import-schema>
02   <xs:schema targetNamespace="http://localhost/ns/yes-no"
03               xmlns:xs="http://www.w3.org/2001/XMLSchema">
04     <xs:simpleType name="yes-no">
05       <xs:restriction base="xs:string">
06         <xs:enumeration value="yes"/>
07         <xs:enumeration value="no"/>
08       </xs:restriction>
09     </xs:simpleType>
10   </xs:schema>
11 </xsl:import-schema>
12
13 <xsl:param name="condition" as="xs:string" select="'yes'"/>
14
15 <xsl:variable name="condition-value" as="my:yes-no"
16               select="my:yes-no($condition)"
17               xmlns:my="http://localhost/ns/yes-no"/>
18
19 ...
20   <xsl:value-of select="$condition-value"/>
```

- the constrained data type using `as=` indicates the type
- the example shows how to constrain a passed parameter using an internally-defined schema:
  - the data type of the passed parameter is the generic simple string `xs:string`
  - the variable casts the parameter value and must be either "`yes`" or "`no`"
  - at run time the processor can confirm the default or overriding value is correct

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Validating result tree nodes

Engaging validation requires:

- using a schema-aware processor
- engaging schema awareness
- specifying in-scope schema definitions in the transformation environment
  - see Importing schema definitions (page 185) for details

Available for document, element and attribute constructors

- user-defined global types from imported schemas
  - e.g. `address:zip-code`
- built-in awareness of XSD and XPath 2 types
  - e.g. `xs:integer`

Two mutually-exclusive attributes for XSLT construction instructions:

- `validation=` strength of the validation of the schema-declared type
  - a value of "`strict`" will perform strict schema validity assessment on element and attribute nodes
  - a value of "`lax`" will perform strict schema validity assessment only on those element and attribute nodes for which there is a declaration
  - a value of "`preserve`" will retain all data type information from elements and attributes when copied from the source tree to the result tree
  - a value of "`strip`" will remove all data type information from elements and attributes and turn off the translation of the string value to the typed value
    - elements become `xs:untyped`
    - attributes become `xs:untypedAtomic`
    - values of elements and attributes become `xs:untypedAtomic` strings
- `type=` strict validation of the specified named type
  - specifies the explicit type regardless of any declared type for the construct

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespace protection

A special concern regarding the use of namespaces:

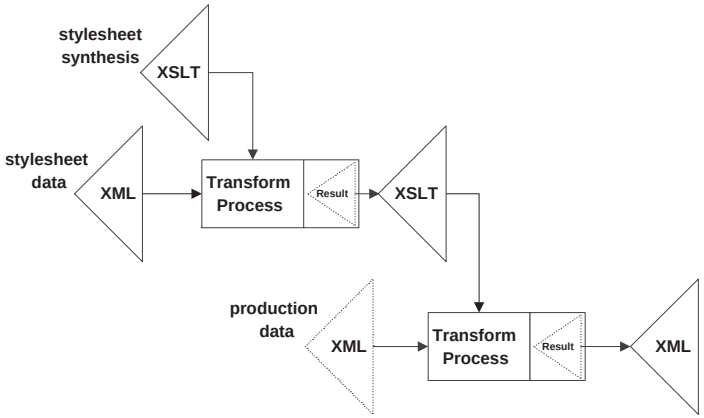
- the "transformation by example" paradigm utilizes literal result elements
  - represent result tree element nodes with associated attribute nodes
    - written as an element with associated attributes in a template in the stylesheet
      - can use the default namespace
      - can use a namespace prefix and associated namespace URI
- some namespaces can be sensitive in the document processing environment
  - automatically-triggered platform services
  - for example: digital signature processing
- if the result tree requires a sensitive namespace, the stylesheet can't use the namespace in a literal result element
  - to produce an XSLT script as the output of translation
    - the XSLT processor would incorrectly interpret the result vocabulary as input
  - to use a platform service for the output of translation
    - the stylesheet use of the URI would incorrectly trigger the service
- `<xsl:namespace-alias stylesheet-prefix="`*`prefix`*`" result-prefix="`*`prefix`*`"/>`
  - top-level element only
  - instruction to translate a namespace prefix in the stylesheet into another namespace prefix when used in the result
  - `stylesheet-prefix=` specifies the prefix used in the stylesheet tree that is being added to the result tree by the stylesheet
    - no influence or recognition in the source tree
  - `result-prefix=` specifies the prefix of the stylesheet tree whose URI is to be used for the result tree prefix
  - the prefix must be declared in the stylesheet even if no element in the stylesheet uses the prefix
  - [2] the value "`#default`" is allowed in XSLT 2.0 to specify the default namespace
- [1] the stylesheet prefix is used in the result tree
- [2] the result prefix is used in the result tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Namespace protection (cont.)

Classic use case: stylesheets writing stylesheets

- the output of one transformation is the stylesheet for another transformation



Note that this is how the reference implementation of ISO/IEC 19757-3 Schematron is implemented

- `http://www.schematron.com`
- the Schematron script is the "stylesheet data"
- the Schematron stylesheet is the "synthesis stylesheet"
- the resulting XML is the set of validation constraints expressed as an XSLT stylesheet
- that synthesized stylesheet is run on production data to produce a validation report

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespace protection (cont.)

Note in the example below how the XSLT namespace URI cannot be used for the declaration for the `xslo` prefix, otherwise the `xslo` prefixed elements would be interpreted as XSLT instructions:

```
01 <?xml version="1.0"?><!--xsl.xsl-->
02 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
03                 xmlns:xslo="any-URI-but-XSLT-URI" version="2.0">
04
05 <xsl:output indent="yes"/>
06
07 <xsl:namespace-alias stylesheet-prefix="xslo"
08                       result-prefix="xsl"/>
09
10 <xsl:template match="/">                              <!--root rule-->
11   <xslo:stylesheet version="2.0">
12     <xslo:template match="/">
13       <html>
14        <p>
15          Stylesheet <xsl:value-of select="."/>:
16          <xslo:value-of select="."/>
17        </p>
18       </html>
19     </xslo:template>
20   </xslo:stylesheet>
21 </xsl:template>
22
23 </xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Namespace protection (cont.)

Chapter 5 - Transformation environment
Section 3 - Serialization

When this particular stylesheet is run with the following data file as source:

```
<stylesheet-data>input file number one</stylesheet-data>
```

the XSLT processor will build the result tree with the "`xsl`" prefix and associated URI for every element that uses the "`xslo:`" prefix in the stylesheet as indicated in the `<xsl:namespace-alias>` instruction, thus using the XSLT namespace and URI when the result tree is serialized as XML markup:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
   <xsl:template match="/">
      <html>
         <p>
          Stylesheet input file number one:
          <xsl:value-of select="."/>
         </p>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

When the generated stylesheet is run with the following data as source:

```
<production-data>Test file number two</production-data>
```

when output will be:

```
<html>
   <p>
      Stylesheet input file number one:
      Test file number two
   </p>
</html>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing the result tree

Chapter 5 - Transformation environment
Section 3 - Serialization

Serialization is creating a system resource out of an abstract result node tree

- recall diagrams starting on page 35
  - each one shows the result tree as a dotted triangle inside of the process being serialized to some external resource outside of the process
- serialization is optionally specified by the transform writer
  - the nodes in the result tree can be the nodes of a subsequent process's source tree
- serialization is optionally supported by the transform engine
  - a processor need not support any serialization at all and would still be considered conformant

`<xsl:output>` top-level element requests serialization

- request to serialize the result tree as a sequence of characters
  - the XSLT processor *may* choose to respect the request, but is not obliged
- all attributes are optional
- 2.0 name="*`declaration-qname`*"
  - gives the set of declarations a name so that it can be referenced
  - useful when producing multiple result trees (page 38)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing the result tree (cont.)

Serialization properties

- all are optionally specified and all are optionally supported
- presented as attribute specifications but this may not be how a vendor requires it
  - e.g. Saxon uses *`name`*`=`*`value`*, not *`name`*`="`*`value`*`"`
- `method="`*`method-indication`*`"`
  - `method="html"`
    - HTML vocabulary and SGML markup conventions
      - empty elements
      - attribute minimization
      - built-in character entity referencing (as defined by HTML)
      - cannot selectively turn on only a subset of the HTML conventions
        - all conventions are used according to common practice
    - considered the default when the document element is `{}html` in any letter case
      - the name of the document element node is HTML (case insensitive)
      - the null namespace URI is used for the name (i.e.: no namespace prefix)
  - `method="xhtml"`
    - XML serialization with recommended XHTML lexical conventions
      - guidelines for empty tags for elements defined to be empty
      - no markup minimization for empty elements not defined to be empty
    - considered the default when the document element is `{http://www.w3.org/1999/xhtml}html` in lower letter case
  - `method="xml"`
    - arbitrary vocabulary and XML markup conventions
      - empty elements
      - built-in character entity referencing
    - the default when the default isn't HTML or XHTML
  - `method="text"`
    - no vocabulary and no lexical or syntactic conventions
      - serializes only the text nodes of every element in the result tree
      - all characters in clear text (no entities of any kind)
    - never the default
  - `method="`*`prefix`*`:`*`processor-recognized-method-name`*`"`
    - `xmlns:`*`prefix`*`="`*`processor-recognized-URI-reference`*`"`
    - lexical and syntactic conventions recognized by the processor
      - arbitrary serialization (out of the scope of W3C) such as a binary serialization for a colloquial vocabulary
    - never the default

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing the result tree (cont.)

Serialization properties (cont.)

- method serializations are standardized in a separate document
  - shared with XQuery
  - `http://www.w3.org/TR/xslt-xquery-serialization/`
- attributes related to the method:
  - `version="`*`numeric-version`*`"`
    - to specify the version of the output method
  - `omit-xml-declaration="yes"` and `omit-xml-declaration="no"`
    - to specify the absence or presence of the XML declaration (if the result tree represents a document entity) or the text declaration (if the result tree represents an external general parsed entity)
  - `standalone="yes"` and `standalone="no"`
    - to specify the presence or absence of a standalone document declaration
  - `doctype-system="`*`system-identifier`*`"`
    - to specify the system identifier to use in the `DOCTYPE` declaration
  - `doctype-public="`*`public-identifier`*`"`
    - to specify the public identifier to use in the `DOCTYPE` declaration
    - requires `doctype-system=` to also be specified if the output method is XML

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Serializing the result tree (cont.)

Serialization properties (cont.)

- attributes related to the method (cont.):
  - `include-content-type="`*`yes-or-no-default-yes`*`"`
    - only applicable to XHTML and HTML serialization
    - a value of "`yes`" will inject or replace an appropriate `<meta/>` element as the first child of the `<head>` element to declare the content type of the page
    - recall the illustrated result output on page 49
  - `escape-uri-attributes="`*`yes-or-no-default-yes`*`"`
    - only applicable to XHTML and HTML serialization
    - applies normalization, percent encoding and character escaping to URI attribute values
  - `normalization-form="`*`token`*`"`
    - only applies to the XML serialization method
    - used to specify the character selection for the Unicode repertoire
    - see `normalize-unicode()` page 288 for details
    - the value "`none`" specifies no normalization will be applied
    - the prefix "`NF`" represents "Normalization Form"
    - the values "`NFC`" and "`fully-normalized`" are standardized in `http://www.w3.org/TR/charmod-norm/`
      - requests canonical decomposition followed by canonical composition
    - the values "`NFD`", "`NFKC`" and "`NFKD`" are standardized in `http://www.unicode.org/unicode/reports/tr15/`
      - `NFD` requests canonical decomposition
      - `NFKD` requests compatibility decomposition
      - `NFKC` requests compatibility decomposition followed by canonical composition
    - an implementation-defined normalization can be specified by using any other token
  - `undeclare-prefixes="`*`yes-or-no-default-no`*`"`
    - only applicable to XML version 1.1 serialization
    - a value of "`yes`" will inject namespace undeclarations on elements not containing the same namespace nodes as its parent

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing the result tree (cont.)

Serialization properties (cont.)

- attributes related to the serialized markup syntax:
  - `indent="`*`yes-or-no-default-based-on-method`*`"`
    - to ask the processor (at its discretion) to indent the result "nicely" with additional white space when using the `xml` method
      - implications for the downstream parsing processes if the white space is considered significant
      - should only use "`yes`" during development/diagnostics and should use "`no`" in production because of the arbitrary amount of white space being added to the result
  - `cdata-section-elements="`*`list-of-element-type-qnames`*`"`
    - white-space-separated list of element types possibly used in the result
    - specifies those result tree elements whose text content is serialized within a `CDATA` section

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Serializing the result tree (cont.)

Serialization properties (cont.)

- attributes related to the encoding:
  - `encoding="`*`encoding`*`"`
    - to request (if supported by the processor) the character set encoding output of the emitted result tree
      - processor can arbitrarily use `encoding="UTF-8"` or `encoding="UTF-16"` for XML or XHTML
    - value should match the `encoding=` pseudo-attribute described by the XML Recommendation for the XML declaration
    - typical values supported by some processors:
      - e.g. `encoding="US-ASCII"`
        - a 7-bit character set with no accented characters or special symbols
      - e.g. `encoding="ISO-8859-1"` (Latin 1)
        - an 8-bit character set historically used for Western European encoding
        - note that some processors support `encoding="Latin1"`
      - e.g. `encoding="ISO-8859-15"` (Latin 9)
        - an 8-bit character set that replaces the currency symbol ("¤") with the Euro symbol ("€")
  - `byte-order-mark="`*`yes-or-no`*`"`
    - to add a Unicode byte order mark at the start of the file
    - for `encoding="UTF-16"` the default is "`yes`"
    - for `encoding="UTF-8"` the default is implementation defined
    - for all other encodings the default is "`no`"
  - `media-type="`*`media-type`*`"`
    - to specify the MIME content type (without specifying the `charset` parameter)
- attribute related to the character-level serialization
  - `use-character-maps="`*`list-of-named-character-maps`*`"`
    - engage character to string replacement during serialization
    - white-space-separated list of `<xsl:character-map>` instructions describing the translation of characters
    - see Character maps (page 199) for details

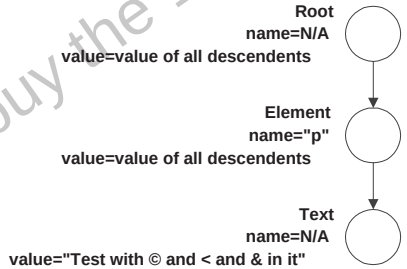
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Illustration of output methods

Consider a simple XML file `nodein.xml` created using the 8-bit ISO character set for Western European languages Latin 1, a.k.a. ISO-8859-1 (note the copyright symbol seen here is encoded in the file using the hexadecimal character `0xA9`):

```
01 <?xml version="1.0" encoding="iso-8859-1"?>
02 <p>Test with © and &lt; and &amp; in it</p>
```

The following illustrates the node tree that is created by the XSL processor:



Note how the markup used to represent the sensitive XML characters is lost. The node tree shown would also be created identically by the following markup:

```
01 <?xml version="1.0" encoding="iso-8859-1"?>
02 <p><![CDATA[Test with © and < and & in it]]></p>
```

All character values in text nodes are maintained as UCS-2 (Universal Character Set - Two Octet) characters. The UCS character set is a 32-bit (4 octet) repertoire with a 16-bit (2 octet) repertoire subset (equivalent to Unicode) that can be serialized as either 16-bit (2 octet) characters or, using an encoding called UTF-8, as a sequence of 8-bit (1 octet) characters.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Illustration of output methods (cont.)

The use of `method="xml"` emits the same nodes using the UCS characters of the text nodes while using the built-in XML entities where necessary:

```
01 <p>Test with © and &lt; and &amp; in it</p>
```

- note the two-character UTF-8 hexadecimal representation of the copyright symbol is `0xC2 0xA9` which would both be revealed in a non-UTF-8 presentation environment such as an ISO-8859-1 Latin-1 environment as follows:
  `<p>Test with Â© and < and & in it</p>`

The use of `method="html"` recognizes known built-in HTML entities and uses the entity references where necessary:

```
01 <p>Test with &copy; and &lt; and &amp; in it</p>
```

The use of `method="text"` ignores all element start and end tags and puts out the UCS characters of all the text nodes while not using any built-in entities:

```
01 Test with © and < and & in it
```

- note again in a non-UTF-8 environment this text file would appear as two characters as in the ISO-8859-1 Latin-1 environment:
  `Test with Â© and < and & in it`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Character maps

1 2 One can declare a set of character translations when the result is serialized

- serialization happens after the result tree is created (see page 35)

Individual Unicode characters are mapped to serialized strings

- each mapping has a mandatory character and a mandatory replacement string:

```
01 <xsl:output-character character="single-Unicode-character"
02                       string="arbitrary-replacement-string"/>
```

- describes a serialization mapping of a single Unicode character in `character=` to an arbitrary string of markup characters in `string=`
  - any Unicode character can be used as the trigger for a string replacement
  - Unicode reserves `&#xFDD0;` to `&#xFDEF;` as non-characters
    - not expected to be used in files by users
    - effectively application-internal private-use code points
  - Unicode reserves `` to `` for private use and does not define any interoperable semantics for these characters
    - but users may be using these private characters for their own use

`<xsl:character-map>` top-level element

- a character map can stand alone

```
01 <xsl:character-map name="character-map-qname">
02   optional <xsl:output-character> elements
03 </xsl:character-map>
```

- a character map can be constructed from other character maps

```
01 <xsl:character-map name="character-map-qname"
02           use-character-maps="white-space-separated-list-of-names">
03   optional <xsl:output-character> elements
04 </xsl:character-map>
```

- the last encountered `<xsl:output-character>` declaration for a particular character is the one used

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Character maps (cont.)

An example of needing to produce ASP results in a DOS-compatible format:

- consider the need to produce "<%" in the result tree when this syntax is not valid HTML or XML use of the "<" character:
  - `<% Method-Call() %>`
- character maps can be set up to convert XML new-lines and to use Unicode reserved characters:

```
01 <xsl:character-map name="dos-format">
02   <xsl:output-character character="&#xA;"
03                         string="&#xD;&#xA;"/>
04 </xsl:character-map>
05 <xsl:character-map name="asp-stuff"
06                     use-character-maps="dos-format">
07   <xsl:output-character character="&#xfdd0;" string="&lt;%"/>

08   <xsl:output-character character="&#xfdd1;" string="%&gt;"/>
09 </xsl:character-map>
```

- the text result then uses the reserved characters to represent the string sequence
  - `&#xFDD0; Method-Call() &#xFDD1;`
- it is not necessary to escape "%>" but one might want to do so for symmetry
- the processor must be directed to utilize the character map
  - `<xsl:output use-character-maps="asp-stuff"/>`

An interesting use case:

- translate accented capital letters to unaccented capital letters
  - France accepts accented capitals without accents, while Québec does not

```
01 <xsl:character-map name="Québec-to-France">
02   <xsl:output-character character="À" string="A"/>
03   <xsl:output-character character="É" string="E"/>
04   ...many others...
05 </xsl:character-map>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Multiple result trees

1 2 One can create more than a single result tree

- each tree can be serialized into a separate sequence of markup characters
- recall page 38

`<xsl:result-document>` instruction

- the children and descendents are serialized into a stream of markup or text
- `href="`*`AVT`*`"` specifies the URI of where to place the resulting serialization
  - this may be a virtual location and not a physical resource of any kind
- `validation=` strength of the validation of the declared type
  - a value of "`strict`" will perform strict schema validity assessment on element and attribute nodes
  - a value of "`lax`" will perform strict schema validity assessment only on those element and attribute nodes for which there is a declaration
  - a value of "`preserve`" will retain all data type information from elements and attributes when copied from the source tree to the result tree
  - a value of "`strip`" will remove all data type information from elements and attributes and turn off the translation of the string value to the typed value
    - elements become `xs:untyped`
    - attributes become `xs:untypedAtomic`
    - values of elements and attributes become `xs:untypedAtomic` strings
- `type=` strict validation of the named type
  - specifies the type of the document element of the result tree
- `format="`*`AVT`*`"` optionally specifies the named `<xsl:output>` element to use as a basis for serialization options
  - omitting this attribute specifies the unnamed `<xsl:output>`
- `output-version="`*`AVT`*`"` optionally overrides `version=` in named `<xsl:output>` instruction
  - this is necessary because for non-empty instructions `version=` refers to the version of the transformation process applicable to the descendent nodes
- all other `<xsl:output>` attributes may be specified
  - see Serializing the result tree (page 191) for details
  - note that the `use-character-sets=` is not an attribute value template
  - all other `<xsl:output>` attributes are attribute value templates in `<xsl:result-document>` even though they are not attribute value templates in `<xsl:output>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Uncontrolled processes

There is no recommendation-based user or stylesheet control over or communication available regarding the following processes implemented by the XSLT processor:

Result Tree Attribute Order:

- the XSLT processor may choose to serialize attribute nodes found in the result tree in any order

Result Tree Serialization Instance Markup:

- the XSLT processor may choose any way it desires to serialize the content of text nodes when the stylesheet does not instruct a given element to be emitted as a CDATA section:
  - using XML built-in character entities for markup-sensitive characters
  - using numeric character entities for markup-sensitive characters or characters not present in the encoding character set
  - using piecemeal CDATA sections
- any original markup syntax from the source file is lost when the source file is abstracted into the source node tree
- other than an entire element emitted as a CDATA section, there is no control available in the stylesheet over which serialization methods are used for text content

Result Tree Construction:

- the stylesheet writer is responsible for dictating the final content of the result tree
- the XSLT processor can use any means to effect the final result as described in the recommendation without necessarily implementing the prose description found therein
- the side-effect free nature of the XSLT design (including the inability to change the value of bound variables) allows an XSLT processor to process portions of the input in parallel and combine the intermediate results into the single final result tree
- the of XSLT allows the processor to choose to not preserve the result node tree when serializing the transformed information to an output instance, thus the result may never actually exist as a complete tree within the processor

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Communication facilities

There are different entities involved when writing and running transforms

- different facilities are available to communicate information between the entities
- the transform itself
- the writer of the transform
- the operator invoking the transform
- the processor/platform on which the transformation is being run
- the schema types available to use in the transform

Static analysis errors

- detected by the processor during the creation of the operation tree
- reported to the operator invoking the transform (hopefully the writer of the transform)

Dynamic evaluation errors

- detected by the processor during the creation of the result tree
- reported to the operator invoking the transform

Type errors

- during either static analysis or dynamic evaluation
- when the type of a value is not correct and cannot be cast to be correct in a given context

P1 No standardized errors or error representation in XPath 1

- each implementation defined their own error identification scheme

P2 Error identification by qualified name

- namespace "`http://www.w3.org/2005/xqt-errors`"
- name structure "`XPST`*`nnnn`*" for static errors
- name structure "`XPDY`*`nnnn`*" for dynamic errors
- name structure "`XPTY`*`nnnn`*" for type errors
- any expanded name can be represented as a URI by making the local name into a local identifier
- e.g. for an XPath syntax error:
  - expanded name: "`{http://www.w3.org/2005/xqt-errors}XPST0017`"
  - qualified name: "`e:XPST0017`" assuming `xmlns:e="http://www.w3.org/2005/xqt-errors"`
  - URI style: "`http://www.w3.org/2005/xqt-errors#XPST0017`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Schema type communication in XSLT 2

Information from the available schemas to the transform:

`type-available(`*`qname-string`*`)`

- returns Boolean value for the presence of the specified name of a schema type
  - returns `true` or `false` to indicate the given type can be successfully referenced by the transform without triggering a dynamic error
  - available simple or complex types can be checked
  - those available as built-in types
  - those available as defined by the implementation
  - some types can only be checked at run time, not at compile time:
    - those available in imported schemas

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Communication facilities in XPath 2

Information from the transform to the operator:

`error()`

- raises runtime error `{http://www.w3.org/2005/xqt-errors}FOER0000`

`error(`*`qname`*`)`

`error(`*`qname?`*`,`*`string`*`)`

`error(`*`qname?`*`,`*`string`*`,`*`item*`*`)`

- raise a runtime error
- returns no value
- the first argument passes to the processor the string comprised of the de-referenced URI of the qualified-name namespace, followed by "`#`", followed by the local name
- the second argument is a user-defined description and is handled in an implementation-dependent fashion
- the third argument is a user-defined error object and is handled in an implementation-dependent fashion

`trace(`*`item*`*`,`*`string`*`)`

- report a message of some kind (e.g. for diagnostics)
- returns the value of the first argument unchanged
- then the first value is converted to a string as in `<xsl:value-of>` and passed with the second argument to the processor to be reported for diagnostic purposes
- the disposition of the information is handled in an implementation-dependent fashion

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Communication facilities in XSLT

Information between the stylesheet and the operator:

- `<xsl:param>`
  - operator to stylesheet
    - invocation-time parameterized value for a globally scoped bound variable
    - the specific mechanism of communication is not standardized
    - the processor may choose to not support value specification
  - a default value can be specified should no value be supplied at invocation
- `<xsl:message>`
  - stylesheet to operator
    - an arbitrary message
      - status of progress
      - content violation
    - the specific mechanism of communication is not standardized
    - the processor may choose to not support relating the message
  - the content is any template (static or calculated)
    - T2 `select=` is an alternative to content
  - `terminate="`*`yes-or-no-AVT`*`"`
    - instruction to stop any further processing of the stylesheet and source files
  - allows the stylesheet to report on semantic validation
    - when content has been detected as being incorrect, messages can report problems to the operator
    - structural well-formedness correctness has already been determined by the XML processor inside the XSLT processor
    - stylesheet could also use XPath to determine structural validity if the XSLT processor does not use a validating XML processor
  - allows the stylesheet to report progress when manipulating large data sets

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Communication facilities in XSLT (cont.)

Information from the processor to the stylesheet:

- obtaining the value of a system property about the XSLT processor:
  - `system-property('`*`prefix`*`:`*`property-name`*`')`
- XSLT namespace indicates reserved system properties:
  - `xsl:version`
    - T1 returns a decimal number of the implementation level of XSLT supported by the processor
    - T2 returns a string value that should be cast to a number for comparison
  - `xsl:vendor` and `xsl:vendor-url`
    - each return a string indicating, respectively, the vendor name and URL
- T2 XSLT 2.0 has additional system properties
  - `xsl:product-name`
    - this should remain constant from one release to the next and across portable platforms
  - `xsl:product-version`
    - this should vary from one release to the next and across non-portable platforms
  - `xsl:is-schema-aware`
    - returns "yes" or "no" based on ability
  - `xsl:supports-serialization`
    - returns "yes" or "no" based on ability
  - `xsl:supports-backwards-compatibility`
    - returns "yes" or "no" based on ability
- other namespaces indicate extension system properties
  - `xmlns:`*`prefix`*`="`*`processor-recognized-URI-reference`*`"`
  - `system-property('`*`prefix`*`:`*`property-name`*`')`
  - the processor returns the empty string for an unrecognized property

Illustration:

- telling the operator the stylesheet uses features not supported by the processor

```
01  <xsl:if test="number(system-property('xsl:version')) &lt; 2.">
02    <xsl:message terminate="yes">
03 Sorry this stylesheet requires XSLT 2.0; Complain
04 to '<xsl:value-of select="system-property('xsl:vendor')"/>'
05 at '<xsl:value-of select="system-property('xsl:vendor-url')"/>'
06 and get a new XSLT processor.
07    </xsl:message>
08  </xsl:if>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Communication facilities in XSLT (cont.)

Chapter 5 - Transformation environment
Section 4 - Communicating with the outside environment

Communication between the stylesheet and the processor (cont):

- 2.0 conditional element inclusion:
  - `xsl:use-when=` on a literal result element
  - `use-when=` on an XSLT instruction
  - a static evaluation of the transformation environment
  - typically used with testing system properties but can test the presence of core and extension functions (but not stylesheet functions)

Protecting a stylesheet from referencing an unavailable function:

```
<xsl:value-of select="t:abc()"
              use-when="function-available('t:abc')"/>
```

An example from the XSLT 2.0 specification:

```
<xsl:include href="module-A.xsl"
     use-when="system-property('xsl:vendor')='vendor-A'"/>
<xsl:include href="module-B.xsl"
     use-when="system-property('xsl:vendor')='vendor-B'"/>
```

- at most one of the inclusions will happen, and possibly neither

Another example from the XSLT 2.0 specification:

```
<xsl:import-schema schema-location="http://example.com/schema"
            use-when="system-property('xsl:is-schema-aware')='yes'"/>

<xsl:template match="/"
              use-when="system-property('xsl:is-schema-aware')='yes'"
              priority="2">
  <xsl:result-document validation="strict">
    <xsl:apply-templates/>
  </xsl:result-document>
</xsl:template>

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

- the first two instructions are absent if the processor is not schema-aware
- note how `<xsl:result-document>` without an `href=` when constructing the result tree is equivalent to using `<xsl:document>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 6 - Transform and data management

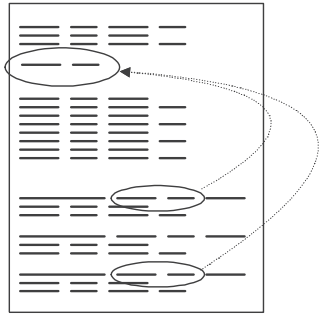
- Introduction - Why modularize logical and physical structures?
- Section 1 - Modularizing the logical structure of transforms
- Section 2 - Modularizing the physical structure of transforms
- Section 3 - Modularizing the source data

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Why modularize logical and physical structures?

Modularizing the logical structure supports development

- manipulating or reusing transform fragments within a given transform
- declaration and reuse of syntactic packages of transform logic and markup
- parameterization of a template
- writing a template once and using it many places
- defining a value and referencing it many places



This chapter overviews logical modularization using:

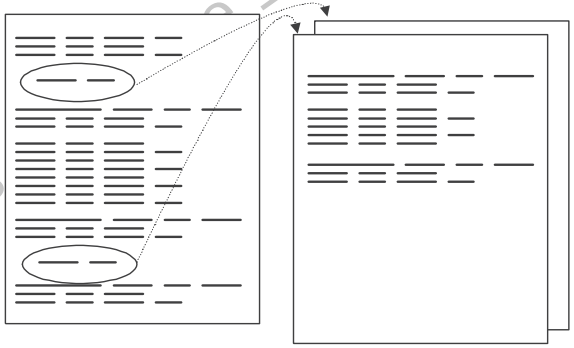
- XML internal general entities in XSLT stylesheets
- XML internal general entities in marked sections in external parameter entities
- variable bindings
- T2 user-defined functions
- XSLT named templates

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Why modularize logical and physical structures? (cont.)

Modularizing the physical structure of transforms supports reuse

- compartmentalization of code
- sharing and reuse of transform fragments across an organization
- support for organizational rules for source code control and management
- access to any built-in custom extension function
  - if available in the processor implementation



This chapter overviews physical modularization using:

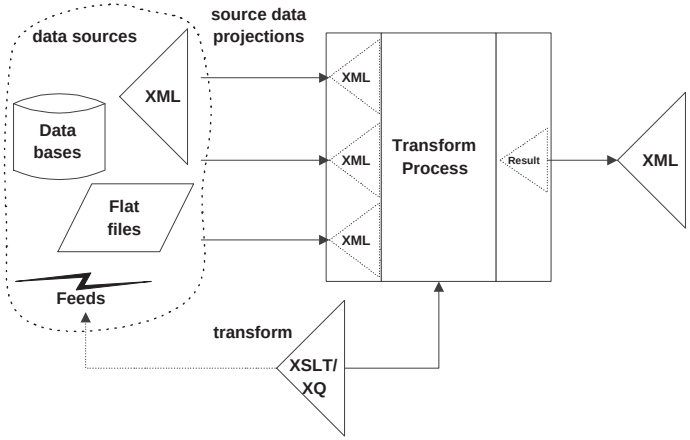
- XML external parsed general entities in XSLT stylesheets
- XSLT included and imported stylesheets
- extension functions
- XSLT extension elements (instructions)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Why modularize logical and physical structures? (cont.)

Modularizing the data supports reuse

- compartmentalization of data
- focusing the responsibility of data to its most appropriate custodians
- sharing and reuse of content across an organization
- accessing content of different kinds through data projection



This chapter overviews physical modularization using:

- XML external unparsed entities
- T2 XPath functions for data access
- XSLT functions for data access

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Why modularize logical and physical structures? (cont.)

Chapter 6 - Transform and data management

The XSLT instructions covered in this chapter are as follows.

Instructions related to logical modularization:

- `<xsl:call-template>`
  - process a stand-alone template on demand
- `<xsl:template>`
  - declare a template to be called by name as an instruction in XSLT
- T2 `<xsl:function>`
  - declare a function to be called by name as an subroutine in XPath
- T2 `<xsl:sequence>`
  - using XPath to express values returned by a function or a template
- `<xsl:variable>`
  - declare a non-parameterized variable and its bound value
- `<xsl:param>`
  - declare a parameterized variable and its default bound value
- `<xsl:with-param>`
  - specify a binding value for a parameterized variable

Instructions related to physical modularization:

- `<xsl:include>`
  - include a stylesheet without overriding stylesheet constructs
- `<xsl:import>`
  - import a stylesheet while overriding stylesheet constructs
- `<xsl:apply-imports>`
  - override the importation of template rules
- T2 `<xsl:next-match>`
  - override the priority and importation of template rules
- `<xsl:fallback>`
  - accommodate the lack of implementation of an instruction element

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Why modularize logical and physical structures? (cont.)

The functions covered in this chapter are as follows.

Availability functions:

- `element-available()`
  - determine the availability of an instruction element
- `function-available()`
  - determine the availability of a function

Functions related to data modularization:

- P2 `collection()`
  - access to a collection of documents
- P2 `doc()`
  - access to multiple source documents
- P2 `doc-available()`
  - check for a document
- `document()`
  - access to multiple source documents
- T2 `unparsed-entity-public-id()`
  - finding the public identifier of an unparsed entity
- `unparsed-entity-uri()`
  - finding the URI of an unparsed entity
- T2 `unparsed-text()`
  - access to multiple documents
- T2 `unparsed-text-available()`
  - check for a document

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Internal general entities

An XSLT stylesheet is an XML document, hence:

- general entities can be declared in the document model
  - defined in the associated schema or Document Type Definition (DTD)
  - in the internal declaration subset
  - in an external file of declarations
- can be used in the body of the stylesheet logic
  - textual replacement allows common sequences of markup to be packaged
  - instructions, content or any text can be defined and re-used
- the use of entities is not preserved
  - the XML processor provides the parsed information to the XSLT processor without distinguishing text encoded with or without entities

```
<?xml version="1.0"?><!--entisamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY Module-word "Module"> <!--teach=module, book=chapter-->
<!ENTITY Lesson-word "Lesson"> <!--teach=lesson, book=section-->
<!ENTITY nbsp " "> <!--known for HTML output, not in XML-->
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="html"/><!--use hardwired browser entities-->
<xsl:template match="/">                        <!-- root rule -->
  <xsl:for-each select="/course/module">
    <p>
      <xsl:text>&Module-word;: </xsl:text>
      <xsl:value-of select="title"/>
      <xsl:apply-templates select="lesson"/>
    </p>
  </xsl:for-each>
</xsl:template>
<!-- ... -->
<xsl:template match="lesson">
  <p>
    <xsl:text>&Lesson-word;: </xsl:text>
    <xsl:value-of select="title"/>
    <xsl:text> - &Module-word;: </xsl:text>
    <xsl:value-of select="../title"/>
  </p>
</xsl:template>
</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Internal general entities (cont.)

The entities can be defined in an external declaration subset rather than the internal declaration subset

- useful for managing a set of entities shared across many stylesheets in a project or an organization

Consider an external parameter entity file "`entisamp2.ent`":

```
<!ENTITY Module-word "Module"> <!--teach=module, book=chapter-->
<!ENTITY Lesson-word "Lesson"> <!--teach=lesson, book=section-->
<!ENTITY nbsp " "> <!--known for HTML output, not in XML-->
```

Pulled into all stylesheets using the document type declaration:

```
<?xml version="1.0"?><!--entisamp2.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet SYSTEM "entisamp2.ent">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
 ...
```

This can help in a strategy of supporting namespace evolution

- declare the namespace URI string in an entity

```
...
<!ENTITY foo-ns "urn:X-Crane:foo:v2">
 ...
```

- use the entity in a namespace declaration

```
...
    xmlns:foo="&foo-ns;"
 ...
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Internal general entities (cont.)

Chapter 6 - Transform and data management
Section 1 - Modularizing the logical structure of transforms

The declarations of internal general entities can be influenced by parameter entity declarations in an external declaration to control the XSLT processor matching of source nodes:

- the XSLT processor acts on the presence of nodes in the source node tree, not on the types of elements found in the document model, hence one can have match patterns for elements never expected to be found
- conditional marked sections are only allowed in external declaration subsets, not in internal declaration subsets, nor in the document element itself
- consider the need to have an XSLT processor behave two different ways for a given element type:
  - can manipulate the element node names being matched by templates in `match=`
  - can manipulate the names of modes in which the templates are collected in `mode=`

Using a conditional marked section with a parameter entity in an external declaration subset, one could have:

```
<!--
    marksamp.ent - External declaration subset example.
-->

<!ENTITY % style2 "IGNORE"> <!--assume without decl means default 1-->
<![%style2;[                  <!--requesting override of 1 with 2-->
  <!ENTITY pane-style-1 "zzz-ignore-pane">
  <!ENTITY pane-style-2 "pane">
]]>
<!ENTITY pane-style-1 "pane"> <!--used if not already defined-->
<!ENTITY pane-style-2 "zzz-ignore-pane">

<!--end of file-->
```

In this example, the value of the `%style2;` parameter entity dictates the value of the two general entities when used in the body of the stylesheet.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Internal general entities (cont.)

Referring to the external declaration subset, one could then have:

```
<?xml version="1.0"?><!--marksamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->

<!DOCTYPE xsl:stylesheet [
<!ENTITY % style2 "INCLUDE">
<!ENTITY % marksamp-ent SYSTEM "marksamp.ent">
%marksamp-ent;
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output method="text"/>

<xsl:template match="&pane-style-1;">
  <xsl:text>Pane Style 1: </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="&pane-style-2;">
  <xsl:text>Pane Style 2: </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Internal general entities (cont.)

The end result reveals the use of the general entities:

```
  Pane Style 2: Pane 1
  Pane Style 2: Pane 2
  Pane Style 2: Pane 3
```

This technique is also useful when wanting to conditionally not process an element by selectively defining the following `&notes-element;` general entity to engage the higher priority template rule, or defaulting with a bogus name to engage the lower priority template rule:

```
<xsl:template match="&notes-element;" priority="1">
  <hr noshade="noshade"/>
  <i>Notes:</i><br/>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="notes"/>                    <!--assume empty-->
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variables and parameters

Variables and parameters are very useful in transforms, they:

- are a value bound to a reference to be reused multiple times without recalculation
  - note the use of the word "variable" is in the mathematical sense of a placeholder or a symbol, not in the traditional programming sense of a storage location that can be modified
- are named using namespace qualified names
  - the default namespace is not used when not using a namespace prefix
  - can prevent inadvertent variable name collision when mixing transform fragments
- are referenced in expressions as: `$`*`variable-qname`*
  - T1 cannot be referenced when the expression is a match pattern
    - template rules
    - key collection declarations
- can be of any type returned by an expression
  - P1 XPath 1.0 data types
    - source tree node set
    - string
    - number
    - boolean
  - T1 XSLT 1.0 result tree fragment data type
    - a direct assignment of rich markup described by the template
      - an empty result tree fragment is treated as an empty string
    - the template can include conditional processing of the template value
  - T2 XSLT 2.0 temporary tree data type
    - temporary trees replace XSLT 1.0 result tree fragments
      - an empty temporary tree is treated as an empty sequence
  - P2 XPath 2.0 sequence types
    - see Sequence types (page 74)
    - any prefix can be used to reference the W3C Schema namespace
    - e.g.: `xmlns:xs="http://www.w3.org/2001/XMLSchema"`
  - P2 user-defined data types
    - from imported schema declarations
- can be validated at compile-time and run-time
  - using `as="`*`sequence-type`*`"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variables and parameters (cont.)

Scope of variables

- when defined with a top-level declaration are globally scoped
  - global scope includes entire import tree, both importing and imported
  - all templates, globally scoped variables and other top-level constructs in all stylesheet fragments regardless of where the fragment is incorporated
- when defined as a parameter of a template or a function are locally scoped
  - any expression within the construct can see the variable
- when defined inside a top-level construct are locally scoped
  - scope is only the following siblings of the declaration and their descendants
  - T1 a local variable does not shadow another local variable
    - a duplicate declaration error is reported if this is attempted
  - T2 a local variable shadows another local variable
- global variables in importing stylesheets shadow other global variables in imported stylesheets
  - not so for included stylesheets
  - described later in the discussion of importation
- local variables shadow global variables

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variables and parameters (cont.)

Classical programming approaches do not apply:

- the theory behind this approach is referred to as side-effect free programming
  - an important implementation constraint that supports parallelism
  - has a robust heritage in LISP-like languages and their derivatives
- e.g. "what is the chapter number?"
  - cannot increment a global variable at the start of each chapter
    - variables cannot vary in their scope once they are bound with a value
  - must ask "what is the current chapter number amongst the chapter siblings?"
  - P1 P2 can use the axes to count nodes
    - using `count()` with the nodes along the sibling axis of an ancestral node
    - e.g.
      `count($n/ancestor-or-self::chapter[1]/preceding-sibling::chapter) + 1`
  - easier in XSLT to let the processor, not the stylesheet, do the counting
    - `<xsl:number>` (see page 391) provides a declarative built-in facility
      - count for the stylesheet writer the quantity of nodes or values required
      - supplants the need to do a lot of "programming" in the stylesheet
      - the act of counting in a stylesheet is awkward when using bound variables as opposed to classical schemes of using global variables
        - may need to use recursively called templates, sometimes with axes
          - a called template introduces a new local scope for variables
        - stylesheets with explicit counting logic can be very verbose and difficult to follow
      - numbering facilities are declarative and easy to follow
        - no need to debug algorithm implementations in stylesheet
    - e.g. `<xsl:number count="chapter"/>`
      - the text of the ordinal number of the closest ancestral chapter (or self)

XSLT has features to process the inherent hierarchical structure of XML

- a classical approach of defining and utilizing variables and array-like structures is more than likely not necessary and is usually far more awkward than a stylesheet approach
- e.g. a classical "programmer's" approach uses `<xsl:number>`, `<xsl:variable>` and `count()` to deal with one's location in the document hierarchy
  - count the current node amongst its siblings using `<xsl:number>` and counting the parent's children using `count()` assigning both to variables and then checking these variables to see where one is in the set of elements
- the stylesheet equivalent approach would be to use `position()` and `last()`
  - process only the elements in question as a current node list and use `position()` and `last()` functions without any variables

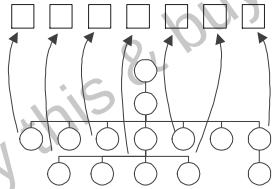
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variable and parameter binding

Very important distinctions regarding result tree fragments and temporary trees

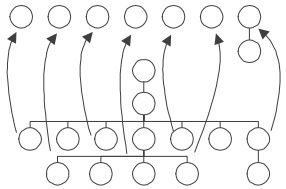
Atomic sequence

```
<xsl:variable name="atom2"
              as="xs:anyAtomicType+"
              select="  expression "/>
<xsl:variable name="atom1"
              as="xs:string+">
 <xsl:for-each select="  expression ">
  <xsl:value-of select="."/>
 </xsl:for-each>
</xsl:variable>
```



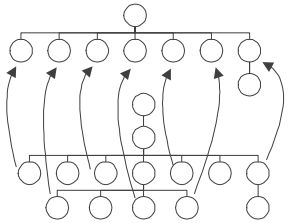
Node sequence

```
<xsl:variable name="seq"
              as="node()+">
 <xsl:copy-of select="  address "/>
</xsl:variable>
```



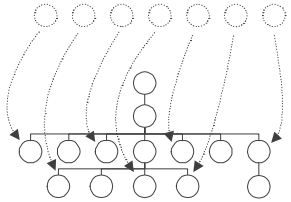
Result tree fragment/ Temporary tree

```
<xsl:variable name="tree">
 <xsl:copy-of select="  address "/>
</xsl:variable>
```



Node set

```
<xsl:variable name="set"
              select="  address "/>
<xsl:variable name="set"
              as="node()+"
              select="  address "/>
```



A node set has only references into the source tree

An atomic sequence is a flat sequence of values without node relationships

A result tree fragment or temporary tree is made of created and copied nodes:

- has a root node regardless of what gets copied or created
- has axes relating the new nodes as siblings that are children of the root

A node sequence is a copy of nodes and their structures

- no root node and copied nodes are not siblings

Recall the sequence types (page 74).

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Variable and parameter binding (cont.)

Binding a value to a variable:

```
<xsl:variable name="variable-qname"
              select="expression-value"/>

<!--XSLT 1.0 empty string; XSLT 2.0 empty sequence if as=-->
<xsl:variable name="variable-qname"/>

<!--XSLT 1.0 result-tree fragment; XSLT 2.0 temporary tree-->
<xsl:variable name="variable-qname">
  sequence-constructor
</xsl:variable>

<!--XSLT 2.0 declaration of a sequence with validation-->
<xsl:variable name="variable-qname"
              as="sequence-type" ...>
```

- specifies the value associated with a given variable's name
- the expression value may be an explicit value or the end result of an expression evaluation
- the content may include any template construction including conditional instructions
  - 1 an empty result-tree fragment is interpreted as an empty string
  - 2 an empty temporary tree value is interpreted as an empty sequence
  - the evaluation of the template in the variable declaration is done a single time and the resulting result-tree nodes are ready to be copied on demand to the result tree without re-evaluation
- the `as=` specifies the precise data type of the variable without any implicit casting
  - 2 a Kleene operator can be used for indicating multiple items are allowed
  -
```
<xsl:variable name="x1">abc</xsl:variable>
<xsl:variable name="x2" as="xs:string">def</xsl:variable>
```
    - `$x1` is a temporary tree, `$x2` is a sequence of one item
  -
```
<xsl:variable name="x3" as="xs:string+">
  <xsl:text>ghi</xsl:text>
  <xsl:text>jkl</xsl:text>
</xsl:variable>
```
    - `$x3` is a sequence of two items
  - 2 the formal declaration of a result tree fragment is as follows:
  -
```
<xsl:variable name="x1" as="document-node()">
  <xsl:document>abc</xsl:document>
</xsl:variable>
```
- unexpected errors as in the following are triggered when `as=` is not precise

```
<xsl:variable name="n-error" select="3" as="xs:positiveInteger"/>
```
  - "3" is of type `xs:integer`, not `xs:positiveInteger`
  - could use `select="xs:positiveInteger(3)"` to cast

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variable and parameter binding (cont.)

A parameter (short for parameterized variable) is identical to a variable except that the expression supplied is considered a default value if an overriding value has not been supplied.

Defaulting a parameterized variable to a value:

```
<xsl:param name="variable-qname"
           select="expression-value"/>

<!--XSLT 1.0 empty string; XSLT 2.0 empty sequence if as=-->
<xsl:param name="variable-qname"/>

<!--XSLT 1.0 result-tree fragment; XSLT 2.0 temporary tree-->
<xsl:param name="variable-qname">
  sequence-constructor
</xsl:param>

<!--XSLT 2.0 declaration of a sequence with validation-->
<xsl:param name="variable-qname"
           as="sequence-type" ...>

<!--XSLT 2.0 mandating a parameter's calling definition-->
<xsl:param name="variable-qname"
           required="yes-or-no-default-no" ...>

<!--XSLT 2.0 indication of an automatically-passed parameter-->
<xsl:param name="variable-qname"
           tunnel="yes-or-no-default-no" ...>
```

- identical in syntax and use to `<xsl:variable>`
- a default value is bound to the parameter when a value is not supplied at invocation
  - top-level parameterized variables are bound at stylesheet invocation
    - an XSLT processor can choose to not support this
  - template parameterized variables are bound at template invocation
- `required=` indicates if a value has to be supplied
  - a static error for explicit calls and a run-time error for command-line invocation
- `tunnel=` indicates if a value is expected to have been automatically passed
  - having such a parameter prevents the need to match the parameters of the calling instruction with those of the called template
  - it is required that this declaration have the attribute in order to access an automatically-passed parameter

An XSLT processor may offer an invocation-time facility to supply values to bind to top-level global parameters

- recall Communication facilities in XSLT (page 206)

Declarations for parameters in a template must precede all other instructions in the template.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variable and parameter binding (cont.)

XSLT notes the semantics of variable assignment are similar to the semantics of Java final local variable declaration with an initialization value, as in:

```
final Object x = "value";
```

as opposed to Java variable assignment that is *not* available in XSLT:

```
x = "value";
```

An example of the direct assignment of a result tree fragment to a variable, then two ways to access the variable is as follows:

```
  <xsl:variable name="test">           <!--variable assignment-->
    <para>test para 1</para>             <!--of rich structure-->
    <para>test para 2</para>
  </xsl:variable>
  <xsl:copy-of select="$test"/>     <!--add fragment to result-->
  <xsl:copy-of select="$test"/>                      <!--again-->
  <value><xsl:value-of select="$test"/></value>  <!--add value-->
```

The resulting markup (two copies and concatenated text node value) is:

```
<para>test para 1</para>
<para>test para 2</para>
<para>test para 1</para>
<para>test para 2</para>
<value>test para 1test para 2</value>
```

Consider the following stylesheet `varsamp.xsl` that reports sample assigned values to variables of each type (note that while in this example the top-level variable assignments follow the reporting in the flow of the stylesheet source, the top-level variables are assigned before the processing of the source node tree begins):

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Variable and parameter binding (cont.)

Chapter 6 - Transform and data management
Section 1 - Modularizing the logical structure of transforms

```
<?xml version="1.0"?><!--varsamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->

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

<xsl:template match="/">              <!--report variables-->
  <root example="{{{$var3a}}} - attribute">
Var1a (result-tree copy-of): {<xsl:copy-of select="$var1a"/>}
Var1a (result-tree value-of): {<xsl:value-of select="$var1a"/>}
Var1b (result-tree copy-of): {<xsl:copy-of select="$var1b"/>}
Var1b (result-tree value-of): {<xsl:value-of select="$var1b"/>}
Var1c (result-tree value-of): {<xsl:value-of select="$var1c"/>}
Var2a (node set copy-of): {<xsl:copy-of select="$var2a"/>}
Var2a (node set value-of): {<xsl:value-of select="$var2a"/>}
Var2b (node set value-of): {<xsl:value-of select="$var2b"/>}
Var3a (string value-of): {<xsl:value-of select="$var3a"/>}
Var3b (string value-of): {<xsl:value-of select="$var3b"/>}
Var3c (string value-of): {<xsl:value-of select="$var3c"/>}
Var4a (number value-of): <xsl:value-of select="$var4a"/>
Var4b (number value-of): <xsl:value-of select="$var4b"/>
Var5a (boolean var1a value-of): <xsl:value-of select="$var5a"/>
Var5b (boolean var1b value-of): <xsl:value-of select="$var5b"/>
Var5c (boolean var1c value-of): <xsl:value-of select="$var5c"/>
Var5d (boolean var2a value-of): <xsl:value-of select="$var5d"/>
Var5e (boolean var2b value-of): <xsl:value-of select="$var5e"/>
Var5f (boolean var3a value-of): <xsl:value-of select="$var5f"/>
Var5g (boolean var3b value-of): <xsl:value-of select="$var5g"/>
Var5h (boolean var4a value-of): <xsl:value-of select="$var5h"/>
Var5i (boolean var4b value-of): <xsl:value-of select="$var5i"/>
Var5j (boolean value-of): <xsl:value-of select="$var5j"/>
Var5k (boolean value-of): <xsl:value-of select="$var5k"/>
  <xsl:text>&#xa;</xsl:text></root>
</xsl:template>
```


Copyright © Crane Softwrights Ltd.
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Variable and parameter binding (cont.)

```
<xsl:variable name="var1a">                    <!--result tree-->
  <xyz attr1="test">This is test 1</xyz>
  <xyz>This is test 2</xyz>
</xsl:variable>
<xsl:variable name="var1b">
  <xsl:apply-templates select="//abc"/>  <!--processing result-->
</xsl:variable>
<xsl:variable name="var1c">
  <xsl:apply-templates select="//def"/>  <!--processing result-->
</xsl:variable>
<xsl:variable name="var2a" select="//abc"/>      <!--node list-->
<xsl:variable name="var2b" select="//def"/>
<xsl:variable name="var3a" select="'string1-val'"/> <!--string-->
<xsl:variable name="var3b" select="''"/>
<xsl:variable name="var3c"/>
<xsl:variable name="var4a" select="42"/>            <!--number-->
<xsl:variable name="var4b" select="0"/>
<xsl:variable name="var5a" select="boolean($var1a)"/><!--bool.-->
<xsl:variable name="var5b" select="boolean($var1b)"/>
<xsl:variable name="var5c" select="boolean($var1c)"/>
<xsl:variable name="var5d" select="boolean($var2a)"/>
<xsl:variable name="var5e" select="boolean($var2b)"/>
<xsl:variable name="var5f" select="boolean($var3a)"/>
<xsl:variable name="var5g" select="boolean($var3b)"/>
<xsl:variable name="var5h" select="boolean($var4a)"/>
<xsl:variable name="var5i" select="boolean($var4b)"/>
<xsl:variable name="var5j" select="true()"/>
<xsl:variable name="var5k" select="false()"/>

<xsl:template match="abc">
  <out attr2="val"><xsl:apply-templates/></out>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Variable and parameter binding (cont.)

When acting on the following simple instance `varsamp.xml`:

```
<?xml version="1.0"?>
<test>Test
<abc test="attr">(abc 1 value)</abc> input.
<abc>(abc 2 value)</abc></test>
```

The stylesheet produces the following result:

```
<?xml version="1.0" encoding="utf-8"?>
<root example="{string1-val} - attribute">
Var1a (result-tree copy-of): {<xyz attr1="test">This is test
1</xyz><xyz>This is test 2</xyz>}
Var1a (result-tree value-of): {This is test 1This is test 2}
Var1b (result-tree copy-of): {<out attr2="val">(abc 1 value)</out><out
 attr2="val">(abc 2 value)</out>}
Var1b (result-tree value-of): {(abc 1 value)(abc 2 value)}
Var1c (result-tree value-of): {}
Var2a (node set copy-of): {<abc test="attr">(abc 1 value)</abc><abc>(abc
 2 value)</abc>}
Var2a (node set value-of): {(abc 1 value)}
Var2b (node set value-of): {}
Var3a (string value-of): {string1-val}
Var3b (string value-of): {}
Var3c (string value-of): {}
Var4a (number value-of): 42
Var4b (number value-of): 0
Var5a (boolean var1a value-of): true
Var5b (boolean var1b value-of): true
Var5c (boolean var1c value-of): true
Var5d (boolean var2a value-of): true
Var5e (boolean var2b value-of): false
Var5f (boolean var3a value-of): true
Var5g (boolean var3b value-of): false
Var5h (boolean var4a value-of): true
Var5i (boolean var4b value-of): false
Var5j (boolean value-of): true
Var5k (boolean value-of): false
</root>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Conditional variable assignment

P2 Conditional constructs in the XPath 2.0 expression

- because of variable scope, the conditional construct cannot go around the declaration
- the conditional construct must go around the declared value

```
<xsl:variable name="use-fee"
   select="if( $discount='yes' ) then (:not as much:) $base*.85
           else if( $discount='internal' ) then (:nada:) 0
           else (:normal amount:) $base"/>
<xsl:if test="$use-fee">
  <p>Fee: <xsl:value-of select="$use-fee"/></p>
</xsl:if>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Conditional variable assignment (cont.)

T1 Conditional constructs in the assignment of a result tree fragment to a variable

- calculating the value of a variable based on criteria
- variables can be declared as result tree fragment
  - converted to required type when being used
- conditional constructs dictate value assigned

Consider the display of monetary values excerpted from `round.xsl`:

```
<xsl:template match="fee">
 <xsl:variable select="ancestor::order/@discounted"
               name="discount"/>   <!--get ancestral attribute-->
 <xsl:variable name="base" select="@base-amount"/><!--get base-->
 <xsl:variable name="use-fee"><!--based on ancestral attribute-->
   <xsl:choose>
     <xsl:when test="$discount='yes'">         <!--not as much-->
       <xsl:value-of select="$base * .85"/>
     </xsl:when>
     <xsl:when test="$discount='internal'"> <!--nothing at all-->
       <xsl:text>0</xsl:text>
     </xsl:when>
     <xsl:otherwise>                          <!--normal amount-->
       <xsl:value-of select="$base"/>
     </xsl:otherwise>
   </xsl:choose>
 </xsl:variable>
<xsl:if test="number($use-fee)">
  <p>Fee: <xsl:value-of select="$use-fee"/></p>
</xsl:if>
</xsl:template>
```

The intuitive (but incorrect) approach would be to put the variable declaration inside the multiple alternative construct:

- the variable's local scope does not extend beyond the encapsulating construct
  - scope is only the declaration's following siblings and their descendants
- the correct approach is to define the variable at the desired scope and bind as the value the result of the alternative construct

Recall the difference between a result tree fragment and an atomic value (page 223)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates

A named template:

- has presence in the node structure of the stylesheet
- is unlike internal entities because of parameterized variables
- has the ability to be invoked (called) from many places
  - with the same or different invocation parameter binding values each time
  - each invocation introduces a new local scope within which variables can have different bound values than used in previously engaged local scopes
- is used to construct the result tree when invoked
- can have parameterized variables
- can have any content
  - nothing at all
  - to simple text
  - to a full result tree template with instructions (including invocations of other named templates)

```
01 <xsl:template name="template-qname">
02   <xsl:param name="variable-qname" select="default-value"/>
```

- gives the template a label to use when being called
- `match=` attribute can also be specified
  - using both allows a template to be used in either or both push and pull approaches as required by the transformation

```
01 <xsl:call-template name="template-qname">
02   <xsl:with-param name="variable-qname" select="overriding-value"/>
```

- constructs the result node tree with the named template
- different than a result tree fragment variable in that the template of a result tree fragment variable is processed once and saved when declared and then copied to the result tree while a named template is processed when called and added "on the fly" to the result tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates (cont.)

Passing a binding value for `<xsl:param>` at invocation time:

```
01 <xsl:with-param name="variable-qname"
02                 select="expression-value"/>
03
04 <!--XSLT 1.0 empty string; XSLT 2.0 empty sequence if as=-->
05 <xsl:with-param name="variable-qname"/>
06
07 <!--XSLT 1.0 result-tree fragment; XSLT 2.0 temporary tree-->
08 <xsl:with-param name="variable-qname">
09   sequence-constructor
10 </xsl:with-param>
11
12 <!--XSLT 2.0 declaration of a sequence with validation-->
13 <xsl:with-param name="variable-qname"
14                 as="sequence-type" ...>
15
16 <!--XSLT 2.0 indication of an automatically-passed parameter-->
17 <xsl:with-param name="variable-qname"
18                 tunnel="yes-or-no-default-no" ...>
```

- parameters are declared with the same kinds of expressions used to declare the values of `<xsl:variable>`

Passing a tunneled parameter makes the parameter available to all nested called templates (not functions)

- the parameter is automatically passed down nested template calls (but not functions) until the process returns from this invocation
- a template's `<xsl:param>` must likewise have `tunnel="yes"` to access automatically-passed parameters

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates (cont.)

Invoking a named template with parameter binding values:

```
<xsl:call-template name="template-qname">
  optional <xsl:with-param> parameter binding values
</xsl:call-template>
```

- supplies values to bind to variables in the called template for its use
- constructs the result node tree with the named template

Applying templates:

- during push one can pass parameter values to the templates that match nodes
  - the values get bound to parameterized variables declared in the template rules invoked by the XSLT processor for each given node selected
  - passing parameter binding values can reduce the amount of calculation performed by an XSLT script if values can be calculated at an early step in the process and passed on
- 1 built-in template rules *do not* support the explicit passing of parameter values in that any passed parameter is not passed along to the templates applied
  - a common problem when passing parameters is to neglect to supply a template rule for a node, thus invoking the appropriate built-in template rule that continues processing but without any passed parameters
- 2 built-in template rules *do* support the explicit passing of parameter values in that every passed parameter is passed along to the templates applied
  - 2 in addition, tunnel parameters are automatically passed, including through built-in templates

```
<xsl:apply-templates select="node-set-expression">
  optional <xsl:with-param> parameter binding values
</xsl:apply-templates>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Named templates (cont.)

Chapter 6 - Transform and data management
Section 1 - Modularizing the logical structure of transforms

2 Rigorous stylesheet writing can catch stylesheet writing errors and runtime errors

- one can declare the kinds of items that are created by the template rule
- if the processor can statically determine the return type is wrong, an error is flagged
- the `as=` attribute indicates how the template result must be structured
- remember the restrictions on types indicated on Sequence types (page 74)
  - but only those for which "`instance of`" can be applied can be used
  - the set of types is smaller when the processor is not schema-aware

Examples:

- the processor will report `test5bad` as being unable to convert the template to a number

```
<xsl:template name="test1" as="xs:decimal">
  5
</xsl:template>

<xsl:template name="test2" as="xs:string">
  <xsl:text>abc</xsl:text>
</xsl:template>

<xsl:template name="test3" as="node()">
  <abc><def/></abc>
</xsl:template>

<xsl:template name="test4" as="node()+">
  <abc><def/></abc><ghi><jkl/></ghi>
</xsl:template>

<xsl:template name="test5bad" as="xs:decimal">
  abc
</xsl:template>
```


Copyright © Crane Softwrights Ltd.
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Named templates (cont.)

A template can return nodes and/or values using XPath instead of XSLT

- use <xsl:sequence select="*expr*"> to return values using XPath
- use content and constructors to return values using XSLT
  - e.g. <xsl:text> and <xsl:value-of select="*expr*"> create text nodes
- recall non-empty <xsl:value-of> (page 150) to return the cardinality of one
  - the content can be any number of constructions
- a sequence selecting a string is less expensive than building and returning a text node
- e.g. <xsl:sequence select="'x'"/> is better than <xsl:text>x</xsl:text>

```
01 <xsl:template name="test6" as="xs:double">
02   <xsl:sequence select="image/@zoom * 10"/>
03 </xsl:template>
04
05 <xsl:template name="test7" as="xs:double?">
06   <xsl:sequence select="image/@zoom * 10"/>
07 </xsl:template>
```

- if the image/@zoom attribute is missing, test6 throws a runtime error and test7 does not

```
01 <xsl:template name="test7bad" as="xs:string">
02   <xsl:text>abc</xsl:text>
03   <xsl:text>abc</xsl:text>
04 </xsl:template>
05
06 <xsl:template name="test8" as="xs:string">
07   <xsl:value-of>
08     <xsl:text>abc</xsl:text>
09     <xsl:text>abc</xsl:text>
10   </xsl:value-of>
11 </xsl:template>
```

- in the first example there are two values being returned which violates the cardinality of 1 declared by the as= constraint
- in the second example, the <xsl:value-of> converts the construction into a single string to satisfy the constraint

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates (cont.)

Declaring a template to be invoked in an XSLT instruction:

- constructs nodes or sequences when matched or when called from an <xsl:call-template>:

```
01 <xsl:template name="template-qname" as="result-data-type"
02               match="XPath-pattern" mode="mode-qname"
03               priority="numeric-priority">
04   optional <xsl:param> parameterized variable declarations
05   optional template content
06   optional <xsl:variable> variable declarations
07   optional template content
08   optional <xsl:sequence> selection using XPath
09 </xsl:template>
```

T1 Processor ignores <xsl:with-param> if conditions are not met

- attempting to pass a binding value for a variable that isn't parameterized
  - when declaring it in the template using <xsl:variable>
- attempting to pass a binding value for a variable that doesn't exist
  - for built-in templates
  - when omitted as an oversight by the stylesheet writer

T2 Processor reports <xsl:with-param> if conditions are not met

- passing a parameter that is not expected is reported as an error

T3 Processor validates resulting sequence against any specified as= data type

- compile-time checking and run-time execution

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates (cont.)

Consider the following input file `ntempsamp.xml`:

```
<?xml version="1.0"?>
<course>
<title>The Course</title>
<module>
<title>The First Module</title>
<lesson><title>Lesson 1 of First Module</title></lesson>
<lesson><title>Lesson 1 of First Module</title></lesson>
</module>
<module>
<title>The Second Module</title>
<lesson><title>Lesson 1 of Second Module</title></lesson>
<lesson><title>Lesson 2 of Second Module</title></lesson>
</module>
</course>
```

Going over the data twice with the following output to be produced:

```
<result>
{The First Module - Lesson 1 of First Module - }
{The First Module - Lesson 1 of First Module - }
{The Second Module - Lesson 1 of Second Module - }
{The Second Module - Lesson 2 of Second Module - }
(The First Module - Lesson 1 of First Module - again! - The Course)
(The First Module - Lesson 1 of First Module - again! - The Course)
(The Second Module - Lesson 1 of Second Module - again! - The Course)
(The Second Module - Lesson 2 of Second Module - again! - The Course)
</result>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Named templates (cont.)

The result is accomplished with the following stylesheet `ntempsamp.xsl`:

```
<?xml version="1.0"?><!--ntempsamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output omit-xml-declaration="yes"/>

<xsl:template match="/">                          <!-- root rule -->
 <result><xsl:text>&#xd;&#xa;</xsl:text>
   <xsl:apply-templates select="//lesson"/>
   <xsl:apply-templates select="//lesson" mode="second"/>
 </result></xsl:template>

<xsl:template match="lesson">       <!--for each lesson first-->
  <xsl:call-template name="lesson-title"/></xsl:template>

<xsl:template match="lesson" mode="second">    <!--second time-->
   <xsl:call-template name="lesson-title">
     <xsl:with-param name="start-delimiter" select="'('"/>
     <xsl:with-param name="end-delimiter" select="')'"/>
     <xsl:with-param name="suffix">
       <xsl:text>again! - </xsl:text>
       <xsl:value-of select="/course/title"/>
     </xsl:with-param>
   </xsl:call-template></xsl:template>

<xsl:template name="lesson-title">      <!--content for suffix-->
  <xsl:param name="start-delimiter" select="'{'"/>
  <xsl:param name="end-delimiter" select="'}'"/>
  <xsl:param name="suffix"/>
  <xsl:value-of select="$start-delimiter"/>
  <xsl:value-of select="../title"/><xsl:text> - </xsl:text>
  <xsl:value-of select="title"/><xsl:text> - </xsl:text>
  <xsl:value-of select="$suffix"/>
  <xsl:value-of select="$end-delimiter"/><xsl:text>
</xsl:text></xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

User-defined functions

1 2 Declaring a function to be invoked in an XPath expression:

- returns nodes or sequences when called from an XPath expression:

```
<xsl:function name="function-qname" as="return-data-type"
              override="yes-or-no-default-yes">
  optional <xsl:param> parameterized variable declarations
  optional sequence content
  optional <xsl:variable> variable declarations
  optional sequence content
  optional sequence constructor
  optional <xsl:sequence> selection using XPath
</xsl:function>
```

A function's name must always be namespace qualified

- to be recognized as a user function in the XPath instruction

A function returns nodes and/or values per a sequence type using `as=` (see page 74)

- engages compile-time and run-time sequence checking

The arity of a function is the count of parameters defined

- there can be any number of functions with the same name but must have different arity
- the arity of the calling reference determines which function definition is invoked
- function parameters are all required and cannot have default values

The `override=` allows a user function to override an extension function

- the default is "`yes`" and ensures consistent behavior across processors
- "no" is useful to make the user function a fallback in a stylesheet running on different processors that have different functions available
  - when the extension function is available, the extension function is used
  - when the function is not available as an extension, the user function is used

No current context defined for "`.`" or "`/`" at invocation

- all XPath location path addresses are invalid until a context is set
  - common to use `<xsl:for-each>` on a passed parameter to set the context
- typically one passes a context as a parameter and uses `<xsl:for-each>` to set the context as being current, on which XPath location path addresses can be used

No access to tunnel parameters

- all parameters must be explicitly passed

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

User-defined functions (cont.)

Note the equivalence of the following four invocations, two being a called template and the other two being function calls, alternately using XSLT and XPath to return values

- there is no constraint on template `thing1`, thus the result is concatenated values

```
  Call: <xsl:call-template name="thing1">
          <xsl:with-param name="p" select="'test'"/>
        </xsl:call-template>
  Call: <xsl:call-template name="thing2">
          <xsl:with-param name="p" select="'test'"/>
        </xsl:call-template>
  Func: <xsl:copy-of select="my:thing3('test')"/>
  Func: <xsl:copy-of select="my:thing4('test')"/>

<xsl:template name="thing1">
  <xsl:param name="p"/>
  <xsl:value-of select="$p"/> <xsl:sequence select="$p"/>
</xsl:template>
<xsl:template name="thing2" as="xs:string+">
  <xsl:param name="p"/>
  <xsl:value-of select="$p"/><xsl:sequence select="$p"/>
</xsl:template>
<xsl:function name="my:thing3" as="xs:string+">
  <xsl:param name="p"/>
  <xsl:sequence select="$p,$p"/>
</xsl:function>
<xsl:function name="my:thing4" as="xs:string+">
  <xsl:param name="p"/>
  <xsl:for-each select="$p">
    <xsl:value-of select="."/><xsl:value-of select="."/>
  </xsl:for-each>
</xsl:function>
```

The end result is as follows:

```
-   Call: testtest
    Call: test test
    Func: test test
    Func: test test
```

Note the impact of the return data type `as="xs:string+"` in the template named "`thing`":

- "`xs:string+`" specifies a sequence of strings
- if the `as=` attribute were omitted, the result would be "`testtest`"

Note in `my:thing3` that "`.`" is undefined until `<xsl:for-each>` sets the context

- this would be true for any other location path relying on nodes
- very common to pass a document context in a parameter and use `<xsl:for-each>` to set the context for the algorithm of the function

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Explicit loop repetition

It is often a requirement to loop a specified number of times to build the result tree

- T1 cannot use an <xsl:for-each> because the number of loops is based on the count of nodes triggered by the select=
  - a template calls itself recursively for the number of additions required to the result tree
    - may require the processor to implement "tail recursion" to avoid the recursive calls exceeding the available stack space
      - tail recursion can only be exploited if there are no components targeted for the result tree after the recursive call in the template
    - each call indicates one fewer repeats are required
- P2 simple use of the "to" operator
  - the expression "1 to $count" returns a sequence of integers inclusive of both operands of the operator
  - <xsl:for-each> will accept a sequence in the select= attribute
    - note that when the context item is an atomic value, there is no definition of "/", nor can any node be addressed without using a variable or function

```
<xsl:text>Dan:   "Say goodnight, Dick."&nl;</xsl:text>
<xsl:for-each select="1 to $count">
  <xsl:text>Dick:  "Goodnight Dick!"&nl;</xsl:text>
</xsl:for-each>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Explicit loop repetition (cont.)

The countdown template rule in the following example is a generic template useful for repetitive copying of any content when invoked

```
<?xml version="1.0"?><!--loop.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [ <!ENTITY nl "&#xd;&#xa;"> ]>

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

<xsl:param name="count" select="3"/>       <!--allow override-->

<xsl:template match="/">
  <xsl:text>Dan:   "Say goodnight, Dick."&nl;</xsl:text>
  <xsl:call-template name="countdown">    <!--begin countdown-->
                <!--convert to number in case supplied as string-->
    <xsl:with-param name="counter" select="number($count)"/>
    <xsl:with-param name="content">  <!--what is to be copied-->
      <xsl:text>Dick:  "Goodnight Dick!"&nl;</xsl:text>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template name="countdown"> <!--recursive loop until done-->
  <xsl:param name="content"/>      <!--structure to be copied-->
  <xsl:param name="counter" select="0"/>   <!--remaining reps-->
  <xsl:if test="$counter > 0">  <!--count not zero; more work-->
    <xsl:copy-of select="$content"/>
    <xsl:call-template name="countdown">   <!--next; one less-->
      <xsl:with-param name="counter" select="$counter - 1"/>
      <xsl:with-param name="content" select="$content"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

External parsed general entities in XML files

An XML external parsed general entity:

- is a stand-alone XML stream of data suitable for referencing from within another XML entity
  - see figure in Extensible Markup Language (XML) (page 9) for an illustration
  - by XML 1.0 rules (section 4.3.2) the content of the entity must be well formed
  - the reference is resolved by the XML processor within the XSLT processor
- can contain any stylesheet fragment residing outside of the stylesheet that invokes it
  - the entity could represent a portion of or an entire template
  - the entity might be synthesized by some pre-process that is used to parameterize the behavior of the stylesheet or the nature of the result markup
- supports management of very large XML files
  - breaking the entire work into manageable fragments
- is a syntactic fragmentation method
  - *not* designed for stylesheet fragment re-use between different stylesheets
  - see `<xsl:import>` and `<xsl:include>` for semantic stylesheet fragment re-use

All boundaries of external parsed general entities are lost

- the XSLT processor cannot distinguish between markup having been found in the main stylesheet entity from markup having been found in an entity outside the main stylesheet entity
- any stylesheet nodes created from the external parsed general entity are added to the stylesheet node tree as if found within the main stylesheet entity itself

Suitable for source files

- users can create data files with any level of such fragmentation
- no change to the stylesheet when changing the fragmentation of a source file

Possible but not suitable for XSLT stylesheets

- stylesheets are XML files but the language design provides better facilities than entities

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Included stylesheets

Semantic inclusion of stylesheets:

- `<xsl:include href="`*`uniform-resource-identifier`*`"/>`
- wholly incorporating another stylesheet
  - the external file itself is a complete XSLT stylesheet with requisite declarations
  - the external XSLT stylesheet may have its own internal declaration subset
    - an internal subset cannot be used in an external file when that file is included as an external general entity
    - the internal subset used in the included file may have own general entities
- included stylesheet nodes added to the stylesheet tree in situ
  - any stylesheet nodes created from the included file are added to the stylesheet node tree as if found within the stylesheet itself at the point of inclusion
- no special accommodation of template conflicts
  - no distinction of included constructs from those in the main stylesheet entity
  - location of inclusion important for error recovery during template conflict resolution

Important role when managing large stylesheets as multiple pieces

- all included fragments are dealt with same level of import precedence
  - one can have multiple template rules accommodating the same node with different levels of priority in different included stylesheet fragments because all included fragments have the same level of import precedence

No polymorphism of global constructs

- duplicate declarations are in conflict and are reported as errors

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Imported stylesheets

Semantic importation of stylesheets:

- `<xsl:import href="`*`uniform-resource-identifier`*`"/>`
- partially incorporating another stylesheet
  - the external file itself is a complete XSLT stylesheet with requisite declarations
    - the internal subset used in the included file may have own general entities
- constructs in importing stylesheet override constructs in imported stylesheet
  - duplicate declarations are not in conflict
- importation and order defines "importance" for template conflict resolution
  - stylesheet constructs in the imported file are considered less important
    - less important than any corresponding constructs in the importing stylesheet
      - thus are not available to be used when there are identically-identified constructs found in the importing stylesheet
    - corollary: importing stylesheet constructs are more important than imported constructs
      - chosen before (without being considered a conflict) any identically-identified constructs found in an imported stylesheet
  - order increases in importance for multiple importation
    - when more than one file is being imported, former importation confers less importance than latter importation
      - corollary: latter imported stylesheet constructs from a given importing file are considered more important
        - chosen before (without being considered a conflict) any identically-identified constructs found in former imported stylesheets
      - importance is transitive
        - all stylesheet constructs from latter imported stylesheets (including those that may be imported into the imported stylesheet) are more important than constructs from prior imported stylesheets

Ideal for tweaking large working stylesheets

- e.g. a new stylesheet needs 95% of an existing stylesheet
  - only 5% written as overriding constructs more important than found in the imported stylesheet
- an acceptable method for encapsulating semantic validation algorithms for re-use by multiple stylesheets

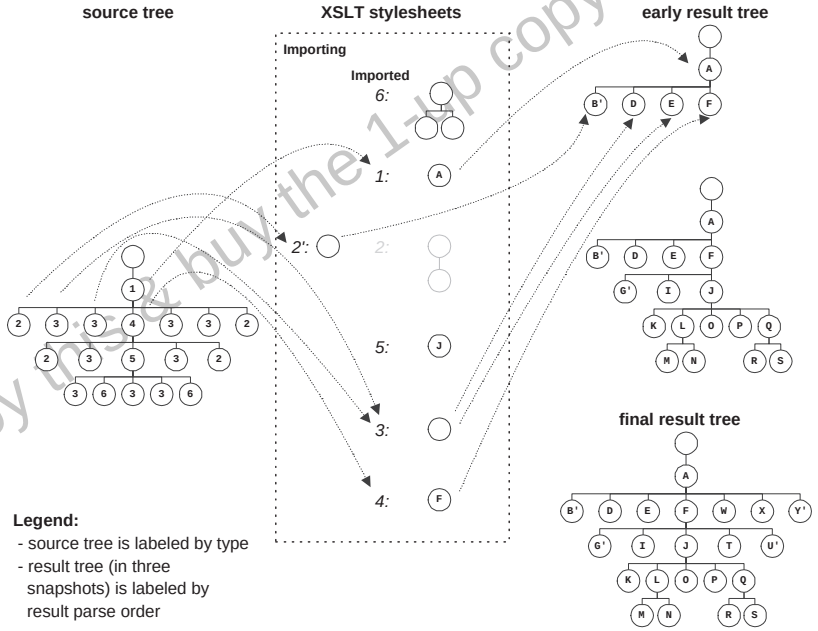
Conference paper regarding using a large import tree in a production system

- see the "Global large-scale stylesheet deployment case study" link from the Crane Softwrights Ltd. home page under "Recommended reading"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Imported stylesheets (cont.)

Importation that supplants imported behaviors:



- the nodes that are pushed to the outer importing stylesheet are intercepted by template rules
- those nodes not intercepted by the importing stylesheet are handled by the imported template rules

Recall the basic behavior of template rules on page 23

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Imported stylesheets (cont.)

An illustration of the transitive nature of importation is as follows:

The order of importance of these files in which a given construct is found is:

1. `file.xsl`
2. `file2.xsl`
3. `file2a.xsl`
4. `file1.xsl`
5. `file1a.xsl`

Note that the above order is exactly the reverse order of parsing the imported files when parsed at the point they are declared, and since all imported files are declared in a stylesheet before any other stylesheet content, this makes it straightforward for implementers to simply override a given construct's definition with a subsequent definition in parse order.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Imported stylesheets (cont.)

Chapter 6 - Transform and data management
Section 2 - Modularizing the physical structure of transforms

Important caveat when managing large stylesheets as multiple pieces

- importance trumps priority
  - one cannot have multiple template rules accommodating the same node with different levels of priority in different imported stylesheet fragments because the priorities of the fragment with the highest importance that matches the node will satisfy the need without looking for higher priority template rules in fragments of lesser importance

Other importation requirements:

- all imported stylesheets must be declared at the start of the importing stylesheet
  - before ***any*** other construct is declared
- imported stylesheets from included stylesheets are imported after imported stylesheets of the including stylesheet
  - all included stylesheets are examined in order, looking for imported stylesheets

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Imported stylesheets (cont.)

Overriding the template match's importance:

- `<xsl:apply-imports/>`
  - useful to supplement existing processing available from imported stylesheets
  - no attributes can be specified
  - T1 no child elements can be specified
  - T2 child `<xsl:param>` elements can be specified
  - acts a lot like `<xsl:apply-templates select="." mode="`*mode-qname*`">` but not quite:
    - does not respect any template rules in the importing stylesheet, only the imported stylesheets
    - does not change the XPath context as would the instruction above
- reapplies the last matched node as if the importing stylesheet didn't exist
  - only uses the template rules of the imported stylesheets of the template rule's stylesheet fragment
- cannot be called within the context of `<xsl:for-each>`

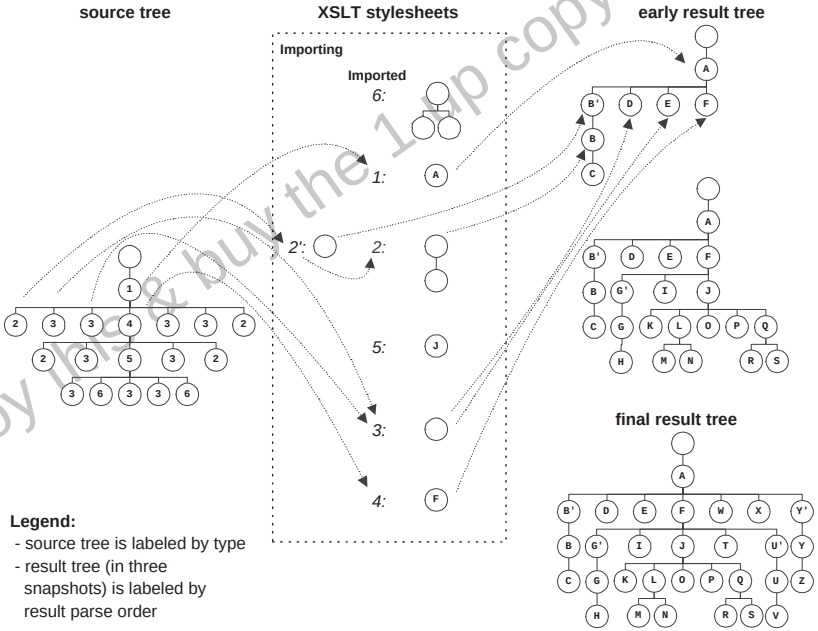
T2 Overriding the template match's importance and priority:

- `<xsl:next-match/>`
  - useful to supplement existing processing available from all stylesheets in the import tree
- reapplies the last matched node as if the matched template didn't exist
  - whichever template matched the last-matched node is removed from the set of templates in play
  - each template that matches and uses another `<xsl:next-match>` is also removed from the set of templates in play
- cannot be called within the context of `<xsl:for-each>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Imported stylesheets (cont.)

Importation that augments imported behaviors:



- the nodes that are pushed to the outer importing stylesheet are intercepted by template rules
- those nodes not intercepted by the importing stylesheet are handled by the imported template rules
- the handling of the importing stylesheet template rule engages the corresponding template rule of the imported stylesheet

Recall the basic behavior of template rules on page 23

Recall the importation behavior that supplants imported stylesheets on page 247

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Imported stylesheets (cont.)

Overriding reprocesses last-matched node

- the context item is not changed
- the current list is not changed
- the mode is not changed

Overriding cannot be triggered in certain instructions

- these instructions cannot be used within the following instructions
  - `<xsl:for-each>`
  - `<xsl:for-each-group>`
  - `<xsl:analyze-string>`
  - `<xsl:sort>`
  - `<xsl:key>`

🔢 Parameters can be passed when overriding:

```
<xsl:apply-imports>
  optional <xsl:with-param> parameter binding values
</xsl:apply-imports>

<xsl:next-match>
  optional <xsl:with-param> parameter binding values
</xsl:next-match>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extension mechanisms

XSLT is extensible in design:

- a transformation can reference functions and instructions not defined by the W3C
- some extension techniques offer fallback definition
  - user-defined actions of what to do when the extension is unavailable
- specified techniques for use
  - how to declare an extension
  - how to invoke an extension
  - how to accommodate a processor that hasn't implemented an extension
- no specified techniques for implementation

Important portability issue

- a processor is *not* required to support any extension at all
- a transform that must be portable across different engine implementations cannot rely on the implementation of extensions mechanisms

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Extension mechanisms (cont.)

Extension functions

Expression evaluated by a custom implementation within the processor:

- called by referencing their namespace-qualified names, optionally providing parameters to be used by the function
  - functions are distinguished by their arity (the number of arguments)
- `xmlns:`*`function-namespace-prefix`*`="`*`processor-recognized-URI`*`"`
- `function-available( '`*`function-namespace-prefix`*`:`*`function-name`*`' )`
- 🅉2 `function-available( '`*`function-namespace-prefix`*`:`*`function-name`*`', `*`arity`*` )`
  - built-in function to determine if the given function is available to be called
    - 🅟2 when passing a numeric value for the arity, determines that a function with the specified number of arguments is available to be called
  - while typically used for extension functions, note that this function can also be used to test the availability of a built-in function defined in the recommendations or a user function:
    - `function-available( '`*`standard-function-name`*`' )`
    - `function-available( '`*`user-function-name`*`' )`
  - returns `false` if function not implemented or available
    - must be available in the function library context of built-in functions, user-defined functions and extension functions
- *`function-namespace-prefix`*`:`*`function-name`*`( )`
- *`function-namespace-prefix`*`:`*`function-name`*`( `*`function-arguments`*` )`
- no fallback available
  - 🅃1 an error is signaled when trying to execute an unimplemented function
  - 🅃2 an error is signaled when trying to reference an unimplemented function

🅃2 Protecting a stylesheet from referencing an unimplemented function:

```
01 <xsl:value-of select="t:abc()"
02               use-when="function-available('t:abc')"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extension mechanisms (cont.)

Extension instruction elements

Instruction evaluated by a custom implementation in an XSLT processor:

- executed by an element with a namespace-qualified element type name
  - may have attributes that may or may not allow attribute value templates
  - considered as instructions and are not considered the same as literal result elements
  - not allowed as top-level elements
    - can only be used as instructions within templates
- `xmlns:`*`instruction-namespace-prefix`*`="`*`processor-recognized-URI`*`"`
- `xsl:extension-element-prefixes="`*`white-space-separated-prefixes`*`"`
  - an ancestral element in the stylesheet must declare the namespace prefix
    - if not declared, stylesheet node is regarded as representing a literal result element
    - `#default` is used to specify the default namespace as an extension element namespace
- `element-available( '`*`prefix`*`:`*`element-type`*`' )`
  - built-in function to determine if a given element type is an instruction
  - note that this function can also be used to test the availability of an XSLT element defined in the recommendation, not just an extension element
    - `element-available(`*`XSLT-prefix`*`:`*`element-type`*`)`
  - returns `false` if instruction not implemented or available
- `<`*`instruction-namespace-prefix`*`:`*`instruction-name`* *`optional-attrs`*`>`
  `optional-instruction-content-and-fallback`
  `</`*`instruction-namespace-prefix`*`:`*`instruction-name`*`>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extension mechanisms (cont.)

Accommodating unimplemented extension instruction elements:

- invocation instantiates `<xsl:fallback>` stylesheet templates in the result tree if instruction not implemented or available
  - immediate child or multiple immediate children of either an XSLT element or an extension element in the stylesheet
    - other descendent fallback templates are not added to the result tree
  - used only when the processor doesn't implement the instruction element
  - ignored when the processor does implement the instruction element
  - all fallback element templates instantiated in order when instruction element executed
  - an error is signaled if there are no fallback child elements when trying to execute an unimplemented instruction element
- useful technique for engaging a comment-like construct around template constructs
  - in XML one cannot use a comment construct around an existing comment construct
  - the combination of an ignored extension element and an empty `<xsl:fallback>` constitutes no activity
    - consider a block of XSLT functionality as follows:

```
01 <xsl:for-each select="a">
02   <!--create the A result-->AAA
03 </xsl:for-each>
04 <xsl:for-each select="b">
05   <!--create the B result-->BBB
06 </xsl:for-each>
07 <xsl:for-each select="c">
08   <!--create the C result-->CCC
09 </xsl:for-each>
```

    - to "comment out" the middle construct, one cannot use a comment but one could use an unimplemented extension instruction with a fallback

```
01 <xsl:for-each select="a">
02   <!--create the A result-->AAA
03 </xsl:for-each>
04 <my:ignore xmlns:my="my-ns"
05             xsl:extension-element-prefixes="my">
06   <xsl:fallback/>
07   <xsl:for-each select="b">
08     <!--create the B result-->BBB
09   </xsl:for-each>
10 </my:ignore>
11 <xsl:for-each select="c">
12   <!--create the C result-->CCC
13 </xsl:for-each>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Modularizing the source data

Modularizing and sharing data fragments is important when strategizing information management

- centralizing the definition of information removes consistency problems introduced when making copies of information
- managing the information by the custodian or owner of the information allows users of the information to focus on other things
  - e.g. not having engineers craft legal prose, let the legal department be responsible for the text used

Using the physical entity structure for sharing data fragments is indelicate

- recall page 9
- external parsed general entities are suitable for fragmenting a large instance into many pieces, but not for sharing the pieces
- each XML document has its own parsing context of defined entities
- changing a referenced external parsed general entity to suit one including instance may make other including instances not well formed
  - due to a reference to an entity in the first instance's parsing context that is not available in the second instance's parsing context

Using separate XML documents for information supports maintenance

- each document is, at the least, XML well-formed
- each document may, itself, be fragmented by using external parsed general entities, but that isn't noticed by the transform accessing the XML document

Transforms can bring into the transformation picture an arbitrary number of node trees

- recall page 212
- processors recognizing different URI strings create node trees from different kinds of sources
- the result can be constructed using nodes from any source node tree

Each transformation product has its own conventions for URI strings for resources

- typical Internet protocols are widely supported
- different conventions to engage different data projections
- understanding the URI schemes available in each processor is *critical*ly important
  - e.g. how to find resources
  - e.g. how to perform queries on data bases

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Unparsed entity referencing in XSLT

Accessing the set of unparsed entity declarations for a document:

- see figure in Extensible Markup Language (XML) (page 9) for an illustration
- the attribute named "`ent`" is pointing to the entity named "`x`" which is pointing to the URI "`x.gif`" declared to be in the notation named "`gif-file`" understood by an application as being of the type described by the URI "`gif-uri`"
- XSLT does not give any access to notation declarations or to the URI associated with a notation name

Unparsed entities are general entities declared in the DTD with an `NDATA` specification of the notation of the entity.

The root node of every document includes a mapping of each unparsed entity to the associated URI:

- the URI may be resolved by the XSLT processor from the public identifier
- otherwise the URI is the system identifier
- when the URI is relative, it is resolved to be absolute by using the base URI of the entity declaration

An XSLT stylesheet can obtain the public and system identifiers for an unparsed entity using:

- document used is that of the current node
- if no such entity exists, the empty string is returned

`unparsed-entity-uri(`*`entity-name-string-argument`*`)`

- returns the specified unparsed entity's absolute system identifier

`unparsed-entity-public-id(`*`entity-name-string-argument`*`)`

- returns the of the specified unparsed entity's public identifier

The file `imgsamp-ent.xml` uses an unparsed entity to point to a graphic's file name, rather than directly encoding the filename in an attribute:

```
<!DOCTYPE imgsamp [
<!NOTATION gif SYSTEM "">
<!ENTITY logo SYSTEM "crane.gif" NDATA gif>
<!ENTITY house PUBLIC "urn:X-Crane:pix:house" "house.gif" NDATA gif>
]>
<imgsamp>
<para>Here is an example:</para>
<fig file="logo">Crane Logo</fig>
<para>And a line drawing of a house:</para>
<fig file="house">House drawing</fig>
<para>End of example</para>
</imgsamp>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Unparsed entity referencing in XSLT (cont.)

The stylesheet `imgsamp-ent.xsl` references the filename to be displayed through the entity declaration found in the DTD of the source file:

```
<?xml version="1.0"?><!--imgsamp-ent.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY dark-green "#004400">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           version="1.0"
           xmlns="http://www.w3.org/TR/REC-html40">

<xsl:output method="html"
         doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"/>

<xsl:template match="/">                    <!-- root rule -->
  <html>
    <head><title>Image Sample</title></head>
    <body><xsl:apply-templates/></body>
  </html>
</xsl:template>

<xsl:template match="fig">
  <div style="font-size:10pt;text-align:center">
    <img src="{unparsed-entity-uri(@file)}" alt="{.}"/>
    <div>
      <xsl:apply-templates/>
    </div>
  </div>
</xsl:template>

<xsl:template match="para">
  <div style="font-size:15pt;color=&dark-green;">
    <xsl:apply-templates/>
  </div>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Document referencing in XPath 2

Chapter 6 - Transform and data management
Section 3 - Modularizing the source data

P2 `doc(`*`uri-string`*`)`
- returns a document node for the apex of the tree referenced by the URI based on the processor's "available documents"
- a run-time error is triggered if a document node is not returned
- if the URI has a fragment identifier, the document-node apex of the tree is the node referenced by the fragment identifier

P2 `doc-available(`*`uri-string`*`)`
- returns `true` if `doc(`*`uri-string`*`)` returns a document node, otherwise returns false

Processors support typical absolute and relative URI strings
- if the URI is a relative URI the base URI is that of the node in the transform fragment where the function is being executed
- e.g. eXist database URI resolution
  - without a protocol, the arguments are considered absolute or relative paths
  - a full URI uses the local host
    - `doc("http://localhost:8080/exist/servlet/db/test.xml")`
  - documented at `http://exist-db.org/xquery.html#N10263`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XPath 2 (cont.)

Chapter 6 - Transform and data management
Section 3 - Modularizing the source data

P2 `collection()` and `collection(`*`uri-string`*`)`
- returns a sequence of nodes based on the implementation's definition of "available collections"
- omitting the string returns the sequence of nodes in the default collection
- e.g. MarkLogic
  - default collection returns the root node of all documents in the database
- e.g. Saxon
  - provides a number of options for selecting files in a collection
  - `collection('./?select=*.xml;recurse=yes;on-error=ignore')`
    - "recursively traverse through directories from the current directory looking for files ending with `.xml`" returning the root node of those that don't fail
      - for an absolute URI in the file system, e.g. the `X:` drive, use `file:///x:/?...`
  - when using the "`file:/`" protocol to point to a directory:
    - the following are query parameters allowed in the URI following "`?`"
    - `select=`*`file-name-pattern`*
      - determines which files in the directory are selected
    - `recurse=`*`yes-or-no`*
      - controls recursively descending the directories
    - `strip-space=`*`yes-or-no`*
      - controls stripping of white-space-only text nodes from the source trees
    - `validation=`*`strict-or-lax-or-preserve-or-strip`*
      - controls the application of validation to the input
    - `on-error=`*`fail-or-warn-or-ignore`*
      - controls the action taken if the file cannot be opened as needed
    - `parser=`*`file-name-pattern`*
      - selects the `XMLReader` Java class to use for reading the files found
  - when using the "`file:/`" protocol to point to a file
    - the file is assumed to be an XML instance listing the document references for the files in the collection
    - ```
      <collection>
        <doc href="doc1.xml"/>
        <doc href="doc2.xml"/>
      </collection>
      ```
  - documented at `http://www.saxonica.com/documentation/javadoc/net/sf/saxon/functions/StandardCollectionURIResolver.html`

Creating the directory list outside the transform is portable:
- `http://code.google.com/p/xml-dir-listing/`
- creates an XML document of directories and their contents
- many features for deciding what goes into the directory listing

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
      ```

Document referencing in XSLT

Accessing documents at run time other than the source document:

`document( non-node-set-uri-string, node-when-not-stylesheet-node )`

- returns the root node of the XML document addressed by the URI string value
  - the URI may include a fragment identifier thus qualifying the node set to be a particular subset of the remote document resource
  - returns the sub-tree of the nodes identified as a standalone tree
- relative URI values are resolved using the base URI of the node value second argument
  - if absent, the stylesheet instruction's Base URI is used

`document( node-set, node-when-not-node-set-node )`

- returns the union of calling `document()` with the string value of each of the nodes
- when the second argument is present, it is used for each call
- when the second argument is absent, uses the Base URI of the node from which the string of the first argument is calculated
  - `document( node )` is different than `document( string(node) )`

`document( "", node-when-not-stylesheet-node )`

- returns the root node of the document fragment in which the supplied node in the second argument is found
- when the second argument absent, the resource is the stylesheet fragment in which the instruction exists that uses the function call

Once the root nodes are obtained from each of the remote documents:

- they can be processed as one would process any node set
  - as either a stand-alone step of a complete location path
  - the first step in a multiple-step location path
- node values can be pulled for inclusion in the result:

```
01 <xsl:value-of select="(document('infofile.xml',.)
02                        //figure)[position()=last()]"/>
```

- nodes can be pushed through the stylesheet for template rule processing:

```
01 <xsl:apply-templates select="document('infofile.xml')
02                              //table"/>
```

Base URI is subdirectory containing the XML syntax for the supplied node

- see Extensible Markup Language (XML) (page 9) for an illustration of external parsed general entities
- stylesheet nodes may come from included or imported stylesheet fragments

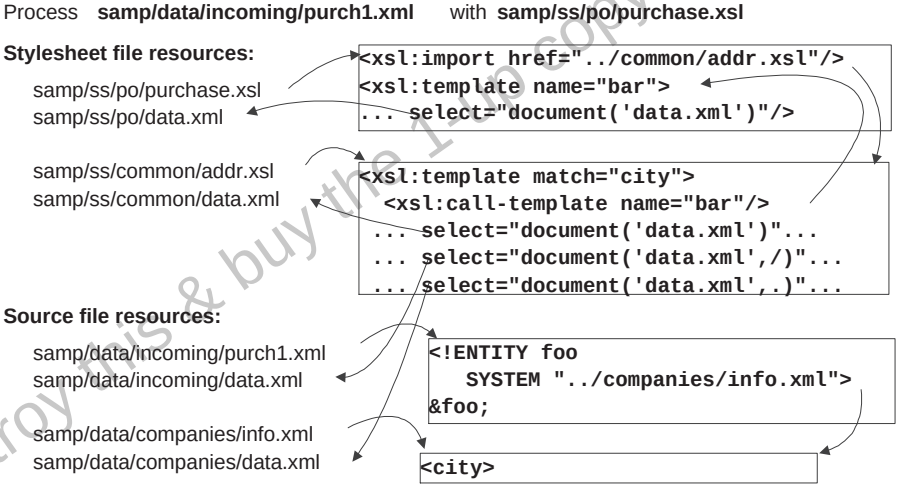
Conference paper regarding wide document retrieval in a production system

- see the "Simple Worldwide Aggregation Using XSLT" link from the Crane Softwrights Ltd. home page under "Recommended reading"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

The following illustrates the possible uses of the `document()` function with a relative URI:



In a file system there are four possible sources of a relatively addressed resource as shown:

- the directory in which the invoked stylesheet file (e.g. `purchase.xsl`) is found
- the directory in which the stylesheet fragment executing the `document()` function (e.g. `addr.xsl`) is found
- the directory in which the invoked source file (e.g. `purch1.xml`) is found
- the directory in which an external parsed general entity including the current node (e.g. `info.xml`) is found

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

Note that with the availability of the stylesheet to this function:

- the stylesheet can contain top-level, non-XSLT constructs containing rich markup that can be accessed as a source node tree (thus providing a location for stylesheet data that is not separated from the stylesheet file itself)
  - note that the container element for such data must have its own namespace as it cannot have the XSLT namespace and it cannot use the default namespace
  - the contents of the container element can use any namespace
  - it may be necessary to use `exclude-element-prefixes=` to keep the data's namespace from being declared in the emitted result tree
- the stylesheet can process result tree fragment top-level variable declarations as a node set as follows:

```
<xsl:apply-templates
    select="document('')/*/xsl:variable[@name='variable-qname']"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

Consider the following example:

```
<?xml version="1.0"?><!--document.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;">]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output method="text"/>

<xsl:template match="/">
  <xsl:text>Processing refs for each document:&nl;</xsl:text>

          <!--using document() as a stand-alone location path-->
  <xsl:for-each select="document(//doc/@loc)">
    <xsl:text>&nl;Resource: </xsl:text>
    <xsl:value-of select="resource/title"/>
    <xsl:for-each select="//ref">
      <xsl:sort/>
      <xsl:text>&nl;</xsl:text><xsl:value-of select="title"/>
    </xsl:for-each>
  </xsl:for-each>
  <xsl:text>&nl;&nl;</xsl:text>
  <xsl:text>Processing references in all documents:</xsl:text>
  <xsl:text>&nl;</xsl:text>

     <!--using document() as the first step in a location path-->
  <xsl:for-each select="document(//doc/@loc)//ref/title">
    <xsl:sort/>
    <xsl:value-of select="."/><xsl:text>&nl;</xsl:text>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

With the control data file `document.xml`:

```
<?xml version="1.0"?>
<set>
  <doc loc="doc1.xml"/>
  <doc loc="doc2.xml"/>
</set>
```

The data file `doc1.xml` with unsorted references:

```
<?xml version="1.0"?>
<resource>
  <title>First Resource</title>
  <ref><title>D - First ref in First resource</title></ref>
  <ref><title>B - Second ref in First resource</title></ref>
  <ref><title>E - Third ref in First resource</title></ref>
</resource>
```

And the data file `doc2.xml` with unsorted references:

```
<?xml version="1.0"?>
<resource>
  <title>Second Resource</title>
  <ref><title>C - First ref in Second resource</title></ref>
  <ref><title>A - Second ref in Second resource</title></ref>
</resource>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

The following result is produced:

```
Processing refs for each document:

Resource: First Resource
B - Second ref in First resource
D - First ref in First resource
E - Third ref in First resource
Resource: Second Resource
A - Second ref in Second resource
C - First ref in Second resource

Processing references in all documents:
A - Second ref in Second resource
B - Second ref in First resource
C - First ref in Second resource
D - First ref in First resource
E - Third ref in First resource
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Document referencing in XSLT (cont.)

Note in XSLT 1 that the `id()` and `key()` functions cannot be used as anything other than the first step in a location path as in the need to reference a unique identifier in another document:

- *it is invalid syntax* to use "`document(argument)/id(argument)`" as a location path
- the context must be changed to the document before locating the context of the unique identifier
- the `<xsl:for-each>` construct effects the repositioning during the instantiation of its template, even when there is only one document to be referenced
- consider the example where one is trying to obtain the value of the "`label`" attribute associated with the element with the unique identifier "`start`" in the document "`otherdoc.xml`":
  - *the incorrect syntax* is:

```
01  <xsl:value-of select="
document('otherdoc.xml')/id('start')/@label"/>
```

  - the correct syntax is:

```
01  <xsl:for-each select="document('otherdoc.xml')">
02    <xsl:value-of select="id('start')/@label"/>
03  </xsl:for-each>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Document referencing in XSLT (cont.)

Chapter 6 - Transform and data management
Section 3 - Modularizing the source data

Accessing non-XML input documents:

- opens the file content as if it were a CDATA section
- but no CDATA limitations preventing "`]]>`"
- returns a text string without parsing the content of the text for any markup
- no end of line normalization is performed
- a relative URI is resolved using the base URI of the instruction with this call

`unparsed-text(`*`uri-string`*`)`

`unparsed-text(`*`uri-string`*`,`*`encoding`*`)`

- the value returned is a string without any interpretation of markup
- quite typical to do string analysis on the retrieved string

`unparsed-text-available(`*`uri-string`*`)`

`unparsed-text-available(`*`uri-string`*`,`*`encoding`*`)`

- returns `true` or `false` based on whether a like call to `unparsed-text()` would be successful

Example to obtain all the lines in a file as a sequence of strings:

- note that because of no end-of-line normalization, this example accommodates end-of-line sequences for DOS, Linux and Mac operating systems

```
01 <xsl:for-each select="tokenize(unparsed-text($infile),'\r?\n|\r')">
02  ...
03 </xsl:for-each>
```

Example to retrieve and put out a text file:

```
01 <xsl:result-document method="text" href="newhello.txt">
02   <xsl:value-of select="unparsed-text('hello.txt')"/>
03 </xsl:result-document>
```

Example to retrieve and put out an HTML file:

```
01 <xsl:result-document method="html" href="newhello.htm">
02   <xsl:value-of select="unparsed-text('hello.htm')"
03                 disable-output-escaping="yes"/>
04 </xsl:result-document>
```

This technique cannot be used for binary files

- the data model can only contain characters that are valid XML characters

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Chapter 7 - Data type expressions and functions

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions

Chapter 7 - Data type expressions and functions

Powerful functions and expression support

- this chapter describes functions and expressions to manipulate variables and values of data types
  - the XSLT specification includes facilities implementing algorithms for publishing-oriented facilities so that the stylesheet writer doesn't have to
- value manipulation
  - boolean functions and operators
  - number functions and operators
  - string functions
  - node set functions and operators
  - P2 sequence functions and operator
  - P2 date and time functions and operators
  - P2 item functions and operators
- P2 regular expressions
- T2 string analysis
- access to the source node tree
  - de-referencing pointers between information items
  - setting up lookup tables to tree nodes

This training material assumes `xmlns:xs="http://www.w3.org/2001/XMLSchema"` when referencing W3C Schema data types

Advanced techniques

- this chapter also describes an approach to walking the source node tree in search of information in such a way that is impossible through available pattern matching techniques.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

The XPath keywords covered in this chapter are as follows.

- `|`
  - union operator
- P2 `union`
  - union operator
- P2 `intersect`
  - intersection operator
- P2 `except`
  - exception operator
- P2 `to`
  - create a sequence of integers
- P2 `instance of`
  - testing the type of an item
- P2 `castable as`
  - testing the conversion of an item
- P2 `cast as`
  - converting an item to the given type
- P2 `treat as`
  - validating an item as a given type
- `or`
  - boolean operator
- `and`
  - boolean operator
- P2 `is << >>`
  - document order comparison operators
- P2 `eq ne lt le gt ge`
  - singleton value comparison operators
- `= != < <= > >=`
  - value comparison operators
- P2 `some every`
  - quantified expressions

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Data type expressions and functions (cont.)

Chapter 7 - Data type expressions and functions

The XSLT instructions covered in this chapter are as follows.

Instruction related to string formatting:

- `<xsl:decimal-format>`
  - control the formatting of numbers when added to the result tree

T2 Instruction related to string analysis and regular expressions:

- `<xsl:analyze-string>`
  - determine the matching components in an analysis of a string
- `<xsl:matching-substring>`
  - act on matching components from the analysis of a string
- `<xsl:non-matching-substring>`
  - act on non-matching components from the analysis of a string

Instruction related to advanced access to the source node tree:

- `<xsl:key>`
  - declare key nodes in the source tree node for bulk processing

Instruction related to advanced algorithmic techniques:

- `<xsl:call-template>`
  - use named templates with subroutine-like control

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Data type expressions and functions (cont.)

The functions covered in this chapter are as follows.

Functions related to boolean data types:

- `boolean()`
  - casting an argument to a boolean value
- `false()`
  - a fixed boolean value
- `lang()`
  - finding the in-scope language as specified by `xml:lang=`
- `not()`
  - inverting the boolean value of the argument
- `true()`
  - a fixed boolean value

Functions related to number data types:

- 🅿2 `abs()`
  - returning the absolute value
- `ceiling()`
  - rounding a number up
- `floor()`
  - rounding a number down
- `number()`
  - casting an argument to a number
- `round()`
  - rounding a number
- 🅿2 `round-half-to-even()`
  - rounding a number

Functions related to string data types:

- 🅿2 `codepoint-equal()`
  - Unicode string comparison
- 🅿2 `codepoints-to-string()`
  - Unicode string conversion
- 🅿2 `compare()`
  - string comparison
- `concat()`
  - string concatenation
- `contains()`
  - string detection

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Functions related to string data types (cont.):

- 🅿2 `default-collation()`
  - obtaining the default collation
- 🅿2 `ends-with()`
  - establish the presence of a string
- `format-number()`
  - adding punctuation and controlling number display
- 🅿2 `lower-case()`
  - string case folding
- 🅿2 `matches()`
  - regular expression matching
- `normalize-space()`
  - normalizing extraneous spaces in a string
- 🅿2 `normalize-unicode()`
  - normalizing Unicode characters in a string
- 🅿2 `replace()`
  - regular expression replacement
- `starts-with()`
  - establishing the presence of a string
- `string()`
  - casting an argument to a string
- 🅿2 `string-join()`
  - join a sequence of strings into a single string
- `string-length()`
  - finding the length of a string
- 🅿2 `string-to-codepoints()`
  - Unicode string conversion
- `substring()`
  - returning a portion of a string
- `substring-after()`
  - returning a portion of a string
- `substring-before()`
  - returning a portion of a string
- 🅿2 `tokenize()`
  - regular expression tokenizing
- `translate()`
  - translating characters found in a string
- 🅿2 `upper-case()`
  - string case folding

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Functions related to sequences:

- P2 `avg()`
  - return the average of members of a numerical sequence
- `count()`
  - return a count of members of the sequence
- P2 `deep-equal()`
  - return an indication of two sequences being identical
- P2 `distinct-values()`
  - return a sequence with duplicate members removed
- P2 `empty()`
  - return an indication of the sequence being empty
- P2 `exactly-one()`
  - return an indication of the cardinality of a sequence
- P2 `exists()`
  - return an indication of the sequence not being empty
- P2 `index-of()`
  - return index pointers into a sequence
- P2 `insert-before()`
  - return a sequence with members inserted
- P2 `max()`
  - return the maximum value of the members of the numeric sequence
- P2 `min()`
  - return the minimum value of the members of the numeric sequence
- P2 `one-or-more()`
  - return an indication of the cardinality of a sequence
- P2 `remove()`
  - return a sequence with a member removed
- P2 `reverse()`
  - return the reverse of a sequence
- P2 `subsequence()`
  - return a portion of a sequence
- `sum()`
  - return a sum of sequence members
- P2 `unordered()`
  - return an unordered sequence
- P2 `zero-or-one()`
  - return an indication of the cardinality of a sequence

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Functions related to date and time:

- P2 `adjust-date-to-timezone()`
  - return adjusted date
- P2 `adjust-dateTime-to-timezone()`
  - return adjusted date and time
- P2 `adjust-time-to-timezone()`
  - return adjusted time
- P2 `current-date()`
  - return date/time component
- P2 `current-dateTime()`
  - return date/time component
- P2 `current-time()`
  - return date/time component
- P2 `dateTime()`
  - return date/time component
- P2 `day-from-date()`
  - return date/time component
- P2 `day-from-dateTime()`
  - return date/time component
- P2 `days-from-duration()`
  - return date/time component
- T2 `format-date()`
  - format date string
- T2 `format-dateTime()`
  - format date and time string
- T2 `format-time()`
  - format time string
- P2 `hours-from-dateTime()`
  - return date/time component
- P2 `hours-from-time()`
  - return time component
- P2 `hours-from-duration()`
  - return date/time component
- P2 `implicit-timezone()`
  - return date/time component

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Functions related to date and time (cont.):

- P2 `minutes-from-dateTime()`
  - return date/time component
- P2 `minutes-from-duration()`
  - return date/time component
- P2 `minutes-from-time()`
  - return date/time component
- P2 `month-from-date()`
  - return date/time component
- P2 `month-from-dateTime()`
  - return date/time component
- P2 `months-from-duration()`
  - return date/time component
- P2 `seconds-from-dateTime()`
  - return date/time component
- P2 `seconds-from-duration()`
  - return date/time component
- P2 `seconds-from-time()`
  - return date/time component
- P2 `timezone-from-date()`
  - return date/time component
- P2 `timezone-from-time()`
  - return date/time component
- P2 `timezone-from-dateTime()`
  - return date/time component
- P2 `year-from-date()`
  - return date/time component
- P2 `year-from-dateTime()`
  - return date/time component
- P2 `years-from-duration()`
  - return date/time component

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Functions related to node data types:

- P2 `base-uri()`
  - obtaining a node's base URI
- P2 `data()`
  - obtaining a node's data
- P2 `document-uri()`
  - obtaining a node's document URI
- `generate-id()`
  - establishing uniqueness in source node trees
- `local-name()`
  - obtaining the local part of a node name
- `name()`
  - obtaining a node name
- `namespace-uri()`
  - obtaining the namespace URI for a node
- P2 `nilled()`
  - obtaining a node's nilled status
- P2 `node-name()`
  - obtaining a node name
- P2 `root()`
  - obtaining the root node of a tree
- P2 `static-base-uri()`
  - obtaining the static base URI

Qualified name functions:

- P2 `in-scope-prefixes()`
  - return a set of prefixes
- P2 `local-name-from-QName()`
  - return a local name
- P2 `namespace-uri-for-prefix()`
  - return a namespace URI
- P2 `namespace-uri-from-QName()`
  - return a namespace URI
- P2 `prefix-from-QName()`
  - return a namespace prefix
- P2 `QName()`
  - return a qualified name
- P2 `resolve-QName()`
  - resolve a qualified name

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data type expressions and functions (cont.)

Other functions:

- `current()`
  - current node access
- 2 `encode-for-uri()`
  - return an encoded string
- 2 `escape-html-uri()`
  - return an encoded string
- `id()`
  - accessing ID values in source node trees
- 2 `idref()`
  - accessing references to ID values in source node trees
- 2 `iri-to-uri()`
  - return an encoded string
- `key()`
  - accessing key nodes from key tables
- 2 `regex-group()`
  - regular expression group retrieval
- 2 `resolve-uri()`
  - return an absolute URI

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using expression functions

Formal function prototypes are in the Recommendations

- only summarized in this material
  - see Annex C Instruction, function and grammar summaries (page 478) for indexed list of functions
  - formal language not copied from Recommendations into this material
  - this material does not attempt to be as rigorous as the Recommendations

Examples of some of the uses of expressions:

`select="`*`expression`*`"`

- evaluation of an arbitrary expression
- selection of nodes for processing

`select="`*`expression`*`,...,`*`expression`*`"`

- evaluation of a sequence of arbitrary expressions

`literal-result-element-attr="{`*`expression`*`}"`

`literal-result-element-attr="{`*`expression`*`,...,`*`expression`*`}"`

- attribute value template calculation
- end result converted to a string and injected into result-tree attribute node
- not applicable for most stylesheet instruction attributes
  - acceptable for specifying the names when constructing named nodes

2 `if (`*`effective-boolean-value-expression`*`)`

`test="`*`effective-boolean-value-expression`*`"`

- used in conditionals
- end result of the expression is converted to its effective Boolean value

`function-name``(`*`function-arguments`*`)`

- a call to a built-in function

`xmlns:`*`function-namespace`*`="`*`URI`*`"`
`...`
`function-namespace``:`*`function-name`*`(`*`function-arguments`*`)`

- a call to an extension function supported by the processor
- 2 a call to a declared user-defined function
- 2 a call to a standard function when using a standard URI
  - e.g. `xmlns:fn="http://www.w3.org/2005/xpath-functions"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using number functions

Double-precision binary floating-point number operators and functions:

- the range of values is defined by 64-bit IEEE Standard for Binary Floating-Point Arithmetic specification (ANSI/IEEE Std. 754-1985):
  - special values are positive and negative infinity, positive and negative zero, "Not-a-Number" (`NaN`)
    - cast, respectively in either `xs:float` or `xs:decimal`, from `'INF','-INF',0,'-0','NaN'`
  - effective Boolean value of positive and negative zero and `NaN` are `false`, all other numbers test `true`

`number(`*`optional-expression`*`)`

- returns the conversion of the string value of the addressed item into an `xs:double` value
- specifying a string argument ignores leading and trailing white space and calculates `NaN` if not valid
- P1 specifying a node set converts the value of the first member of the node set (in document order) to a string before converting to a number
- P2 specifying a node set of more than a singleton is an error
- not specifying an argument converts the current node to a string before converting to a number
- specifying a Boolean argument converts `true` to 1 and `false` to 0

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using number functions (cont.)

Lexical space includes insignificant zero digits

- `0001.` = `1.` = `1.000` = `001.00`
- to the left of significant digits to the left of the decimal separator
- to the right of significant digits to the right of the decimal separator
- P2 E-notation allowed: e.g. `12.345E6` is equivalent to `12345000.`
  - "`E6`" represents "10 to the power of 6" which is 1000000.
- reserved lexical strings for special values: `'INF','-INF','-0','NaN'`

P2 Comparison space distinction between different types of numbers

- types defined in W3C Schema data types `http://www.w3.org/TR/xmlschema-2/`
- augmented by XPath 2.0
  - see Data types (page 73)
- `xs:double` - 64-bit binary floating point (same comparison space as XPath 1.0) IEEE
  - positive and negative numbers with just over 300 powers of 10 both positive (whole) and negative (part)
- `xs:float` - 32-bit binary floating point IEEE
  - positive and negative numbers with just over 37 powers of 10 both positive (whole) and negative (part)
- `xs:decimal` - a decimal number of at least 18 digits (typically many more)
  - no precision problems with a decimal as there are with binary representations
    - e.g. `xs:double(50.2) div 10` returns `5.0200000000000005`
    - e.g. `xs:decimal(50.2) div 10` returns `5.02`
- `xs:integer` - a whole number (no decimal point) with implementation-defined limits
- `xs:long` - an integer from -9223372036854775808 up to 9223372036854775807 (19 digits) = ±2**63
- `xs:int` - an integer from -2147483648 up to 2147483647 = ±2**31
- `xs:short` - an integer from -32768 up to 32767 = ±2**15
- `xs:byte` - an integer from -128 up to 128 = ±2**7
- `xs:nonPositiveInteger` - an integer that is zero or negative
- `xs:negativeInteger` - an integer that is negative
- `xs:nonNegativeInteger` - an integer that is zero or positive
- `xs:positiveInteger` - an integer that is positive
- `xs:unsignedLong` - an integer from zero up to 18446744073709551615 = 2**64 - 1
- `xs:unsignedInt` - an integer from zero up to 4294967295 = 2**32 - 1
- `xs:unsignedShort` - an integer from zero up to 65535 = 2**16 - 1
- `xs:unsignedByte` - an integer from zero up to 255 = 2**8 - 1

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using number functions (cont.)

operators "+", "-", "*"

- traditional arithmetic operations for addition, subtraction and multiplication

operators "div" and "mod"

- traditional division operations of division and modulus
  - the slash "/" cannot be used for division as it is a location path separator in the XPath grammar
- modulus is *not* implemented the same as the IEEE remainder operation
- division by zero is a runtime error

P2 operator "idiv"

- integer cast after performing traditional division operation
- truncates fractional part (no rounding)
- division by zero is a runtime error

P2 operator "to"

- returns a sequence of integers inclusive of both operands of this operator
- e.g. "1 to 3" returns the sequence "(1, 2, 3)"
- ```
  <xsl:text>Dan:   "Say goodnight, Dick."&nl;</xsl:text>
  <xsl:for-each select="1 to $count">
    <xsl:text>Dick (</xsl:text>
    <xsl:value-of select="."/>
    <xsl:text>):  "Goodnight Dick!"&nl;</xsl:text>
  </xsl:for-each>
  ```
- remember to cast using xs:integer(*value*) when not using integers

grouping (and)

- performs the traditional grouping to override precedence

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Calculating values using number functions (cont.)

Chapter 7 - Data type expressions and functions
Section 2 - Number expressions

P2 abs(*number*)

- returns the absolute value of the given number
- e.g. abs(-1.23) returns 1.23
- e.g. abs(4.56) returns 4.56
- the data type of the return value is the base data type of the argument
  - one of xs:float, xs:double, xs:decimal or xs:integer

floor(*number*) and ceiling(*number*)

- returns closest whole number towards negative and positive infinity (respectively)
- e.g. floor(4.9) returns 4.
- e.g. floor(-4.9) returns -5.
- the data type of the return value is the base data type of the argument
  - one of xs:float, xs:double, xs:decimal or xs:integer

round(*number*)

- returns the closest whole number, except in the following situations:
  - the arguments NaN, positive infinity, negative infinity, positive zero and negative zero all return the argument as the result
  - an argument between -0.5 and zero returns negative zero
  - an argument equidistant to two whole numbers (e.g. *x*.5) returns the ceiling() value (whole number closest to positive infinity)
- the data type of the return value is the base data type of the argument
  - one of xs:float, xs:double, xs:decimal or xs:integer

P2 round-half-to-even(*number*) and round-half-to-even(*number*,*precision*)

- returns the number with the closest digit at the specified level of precision
- an argument equidistant to two whole numbers (e.g. *x*.5) returns the closest even whole number
- when precision is supplied, it is the closes whole number to the power of 10 of precision where positive precision is to the right of the decimal point, and negative precision is to the left of the decimal point
  - e.g. round-half-to-even(123456.789, -3) returns 123000.
  - e.g. round-half-to-even(123456.789, 2) returns 123456.79
  - e.g. round-half-to-even(123456.785, 2) returns 123456.78
  - e.g. round-half-to-even(123456.794, 2) returns 123456.79
  - e.g. round-half-to-even(123456.795, 2) returns 123456.8
- when used in rounding many numbers this approach leads to a more balanced distribution rather than always rounding .5 up to the next number
  - approximately half the time it will round .5 down to the previous number
- the data type of the return value is the base data type of the argument
  - one of xs:float, xs:double, xs:decimal or xs:integer

Copyright © Crane Softwrights Ltd.
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.
  ```

Calculating values using number functions (cont.)

Note that when dealing with floating point, sometimes results are not precise; consider the following empirical evidence from `round.xsl`:

```
<xsl:value-of select="format-number(round($use-fee*1.15*100),
                                    '#0.00')"/>
```

- a `$use-fee` value of 8.50 displays with taxes as "9.77"
- the value displayed should be "9.78" because 8.5 * 1.15 * 100 = 977.5 and the `round()` function rounds a fraction of ".5" towards positive infinity
- the floating point library calculated 8.5 * 1.15 * 100 = 977.4999999 which, when rounded to the closest whole number, rounds to 977

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using string functions

String functions:

- strings are sequences of Unicode (a.k.a. Universal Character Set (UCS)) characters
  - lexical space and comparison space are the same (not considering markup)
  - e.g. the value of `select="'abc<def'"` is "`abc<def`"
- separate code points distinguish fully-formed from composite characters
  - composite characters create visual combinations by using non-spacing diacritic marks
  - e.g. the sequence `U+0065 U+0301` ("`e´`") is different than but appears visually the same as `U+00E9` ("`é`")
- P2 a sequential pair of either string delimiter represents a single character
  - e.g. the value of `select='"abc""def&apos;ghi"'` is "`abc"def'ghi`"
  - e.g. the value of `select="'rst''uvw&quot;xyz'"` is "`rst'uvw"xyz`"
  - e.g. the value of `select="'abc&#34;&#34;def'"` is "`abc""def`"
    - only the string delimiter reduces when doubled, not the attribute delimiter
  - e.g. the expression `select="'abc&apos;def'"` is in error
    - the single apostrophe prematurely ends the string definition
  - note that the attribute specification has its delimiters and the value inside the attribute is a string with its own delimiters
  - XML rules do not allow an attribute specification's delimiters to be used inside the specified attribute's value without being escaped
- effective Boolean value of an empty string is `false`, all other strings test `true`

`string(`*`optional-expression`*`)`

- converts the argument to a string
- for any function expecting a string argument, that argument is converted to a string through the use of this function
- specifying an empty node set as an argument returns the empty string
- P1 specifying a non-empty node set as an argument returns the conversion of the first member of the node set (in document order)
- P2 specifying a non-empty node set of more than one member is a runtime error
- specifying a number argument converts the number to sequence of characters
  - note that the result of performing arithmetic very rarely results in expected decimal-place rounding
  - adding two currency amounts with only two digits of significance could produce a floating point number with 20 digits of decimal significance
- specifying a result-tree fragment returns a concatenation of all text nodes in the fragment
- not specifying an argument converts the current node to a string
- specifying a boolean argument converts `true` to the string "`true`" and `false` to the string "`false`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Calculating values using string functions (cont.)

P2 `normalize-unicode(`*`string`*`)` and `normalize-unicode(`*`string`*`,`*`string`*`)`

- returns a string of the supplied string according to either the "`NFC`" method or the supplied normalization method
  - e.g. `NFC` on `U+0065 U+0301` ("`e´`") creates `U+00E9` ("`é`")
    - canonical decomposition followed by canonical composition
- other values can also be specified for the second argument:
  - e.g. `NFD` on `U+00E9` ("`é`") creates `U+0065 U+0301` ("`e´`")
    - canonical decomposition
  - e.g. `NFKD` on `U+2122` ("`™`") creates `U+0054 U+004D` ("`TM`")
    - compatibility decomposition
  - e.g. `NFKC` on `U+1EA3` (LATIN SMALL LETTER A WITH HOOK ABOVE) creates same `U+1EA3` going through `U+0061 U+0309`
  - e.g. `NFKC` on `U+1E9B` (LATIN SMALL LETTER LONG S WITH DOT ABOVE) creates different `U+1E61` (LATIN SMALL LETTER S WITH DOT ABOVE) going through `U+0073 U+0307`
    - compatibility decomposition followed by canonical composition
- the prefix "`NF`" represents "Normalization Form"
- a possible application of this function could be in indexing where unrecognized characters do not participate in the sort
  - recognized characters from decomposing the original characters would participate in the sort

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using string functions (cont.)

P2 Introduction of "collations" used to specify different kinds of comparisons of strings that might be supported by a processor

- e.g. a collation where "ß" (German sharp-s) is ordered between the letters "`s`" and "`t`"
  - any language will do, including English
- use `xsl:default-collation="`*`uri`*`"` to specify collation
- when not specified, the processor supports the Unicode Codepoint Collation `http://www.w3.org/2005/xpath-functions/collation/codepoint`

P2 Collation URI strings are implementation defined

- the Saxon processor recognizes a URI string with the following syntax:
  - ref: `http://www.saxonica.com/documentation/extensibility/collation.html`
  - e.g.: `http://saxon.sf.net/collation?name=value;name=value`
    - `lang=` any value allowed by `xml:lang=`
  - e.g. `http://saxon.sf.net/collation?lang=fr`
    - collate according to the French language conventions
  - strength varies by language by typified by three kinds of differences:
  - `strength=primary` implies `A=a`
    - case insensitive, accent insensitive
  - `strength=secondary` implies `A!=a` and `a=â`
    - case sensitive, accent insensitive
  - `strength=tertiary` implies `a!=â`
    - case sensitive, accent sensitive
  - alphanumeric partitioning grouping sequences of ASCII digits
  - `alphanumeric=yes` (e.g. where "`A234`" is greater than "`A34`")
  - Saxon also supports pointing to arbitrary Java classes with custom code
- MarkLogic URI specifications through pseudo subdirectories:
  - `http://marklogic.com/collation/`*`locale`*`[/`*`attribute`*`]*`
  - see the search developer's guide: `http://developer.marklogic.com/pubs/4.1/books/search-dev-guide.pdf`
    - "Encodings and Collations" chapter

P2 `default-collation()`

- returns the default collation currently in play
- when not specified the Unicode code point collation is used
  - `http://www.w3.org/2005/xpath-functions/collation/codepoint`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using string functions (cont.)

P2 `compare(`*`first`*`,`*`second`*`)` or `compare(`*`first`*`,`*`second`*`,`*`collation`*`)`
- returns integer `-1`, `0` or `1` based on the first argument being less than, equal to, or greater than the second argument when using the optional third argument indicating a collation

P2 `codepoint-equal(`*`first`*`,`*`second`*`)`
- returns boolean `true` or `false` if the first argument is equal to the second argument when using the Unicode code point collation
- the code points are the character references in the Unicode specification
  - `http://www.unicode.org/unicode/standard/versions/`

P2 `codepoints-to-string(`*`integer-sequence`*`)`
- returns a string from a sequence of integer Unicode code points
- e.g. `codepoints-to-string( 65 )` returns `"A"`
- e.g. `codepoints-to-string( ( 65, 66, 67 ) )` returns `"ABC"`
  - note that `codepoints-to-string( 65, 66, 67 )` is in error because that expresses three arguments, not one
- the resulting string is not allowed to contain non-XML 1.0 characters (e.g. control characters from decimal 00 through 31 are disallowed, allowing 9, 10 and 13 for tab, linefeed and carriage return)

P2 `string-to-codepoints(`*`string`*`)`
- returns a sequence of integer Unicode code points from a string
- e.g. `string-to-codepoints( 'abc' )` returns `( 97, 98, 99 )`

`concat(`*`string`*`,`*`string`*`,`*`optional-strings`*`)`
- returns the concatenation of each of the arguments' string value
- requires at least two arguments to be passed to the function
- `concat('abc','def')` returns `"abcdef"`

P2 `string-join(`*`string-sequence`*`,`*`string`*`)`
- returns the concatenation of the strings in the sequence of the first argument, separated by the string in the second argument
- the separator string can be any empty or non-empty string and is not restricted to a single character
- `string-join( ('abc','def'), '' )` returns `"abcdef"`
- `string-join( ('abc','def'), ', ' )` returns `"abc, def"`
- `string-join(('abc','def',''), '
')` separates all strings with a newline
  - useful for text output
  - when the last line needs to end with a newline, add an empty string to the end of the sequence being joined

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using string functions (cont.)

`normalize-space(`*`optional-expression`*`)`
- will act on the value of the current node (if no argument is supplied) or on the argument (if one is supplied)
- strips leading and trailing white space characters and replaces contiguous sequences of adjacent white space characters with a single space character
  - this mimics the same normalization that XML processors do with tokenized attribute values
  - e.g. `normalize-space( ' abc def  ghi ' )` returns `"abc def ghi"`
- handy for checking a node's value having only white space characters
  - e.g. consider that a person entering information may think the following is "empty":
    ```
    <name>
    </name>
    ```
    - the conditional `test="name=''"` is false
    - the string value of the node is not empty because it has a linefeed
    - the conditional `test="normalize-space( name )=''"` is true

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Calculating values using string functions (cont.)

`contains(`*`first`*`,`*`second`*`)` and `starts-with(`*`first`*`,`*`second`*`)`

- returns boolean `true` or `false` if the first argument wholly contains and begins with the complete second argument

P2 `ends-with(`*`first`*`,`*`second`*`)`

- returns boolean `true` or `false` if the first argument wholly contains and ends with the complete second argument

`string-length(`*`optional-argument`*`)`

- returns the count of characters in the supplied string or, if not supplied, in the value of the current node
- "déjà vu": `string-length('déjà vu')` returns 7
- "déjà vu": `string-length('de´ja` vu')` returns 9

`substring(`*`first-as-string`*`,`*`second-as-number`*`,`*`optional-third-as-number`*`)`

- returns the sequence of characters of the first argument from the numeric position indicated in the second argument (1-origin position) to either the end of the string or the count of characters indicated by the number in the third argument (whichever is less)
- `substring( 'abcdef', 4, 2 )` returns `"de"`
- `substring( 'abcdef', 4 )` returns `"def"`

`substring-before(`*`first`*`,`*`second`*`)` and `substring-after(`*`first`*`,`*`second`*`)`

- returns the empty string if the first argument doesn't wholly contain the second argument, or the portion of the first argument that occurs before or after the complete second argument
- `substring-before( 'abc def ghi', ' ' )` returns `"abc"`
- `substring-after( 'abc def ghi', ' ' )` returns `"def ghi"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using string functions (cont.)

P2 `upper-case(`*`string`*`)` and `lower-case(`*`string`*`)`

- returns the upper- or lower-case translation of the string argument
- follows the appropriate case mappings of the Unicode specification
  - `http://unicode.org/Public/UNIDATA/UCD.html`
- does not follow any particular cultural expectation
  - e.g. upper case "é" is either "`E`" in France or "`É`" in Québec
  - the Unicode character database indicates it is "`É`"

`translate(`*`first`*`,`*`second`*`,`*`third`*`)`

- returns the characters of the first argument translated by replacing those characters listed in the second argument with the corresponding characters in the same ordinal position found in the third argument
- characters in the second argument positioned beyond the length of the third argument are removed from the first argument
- `translate( "abcDEFabcDEF", "cDaE", "CdA" )` returns `"AbCdFAbCdF"`
- `translate( "$1,234.56", "$,", "" )` returns `"1234.56"`
  - suitable for numeric comparisons
- `translate( "1 234,56€", ",&#x20ac; ", "." )` returns `"1234.56"`
  - suitable for numeric comparisons
- `translate($value,translate($value,'0123456789',''),'')`
  - returns the digits found in the string value of the context item
  - the inner `translate()` creates a string of every character except digits
  - the outer `translate()` removes those characters thereby leaving only the original digits
-
```
<!DOCTYPE xsl:stylesheet [
<!ENTITY lower 'abcdefghijklmnopqrstuvwxyz'>
<!ENTITY upper 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'>
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
...
    translate($value,'&lower;','&upper;')
```
  - the example converts the `$value` string to upper case
  - suitable for case-less comparisons and presentations
- also could use variables with the case strings instead of entities
-
```
<xsl:variable name="lower" select="'abcdefghijklmnopqrstuvwxyz'"/>
<xsl:variable name="upper" select="'ABCDEFGHIJKLMNOPQRSTUVWXYZ'"/>
...
translate($value,$lower,$upper)
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Decimal formatting in XSLT

`format-number(`*`first-as-number`*`,`*`second-as-string`*`,`*`optional-third-as-string`*`)`

- returns the number represented in the first argument as a string formatted according to the string in the second argument following the decimal format named by the third argument (or the unnamed decimal format if there is no argument)
- the second argument is a string of characters made up of symbols in one or two patterns that dictate how the digits of the positive or negative values of the number are presented in the resulting string
- `format-number( .2, ".00" )` returns `".20"`
- `format-number( .2, "0.00" )` returns `"0.20"`
  - the zero-digit indication displays digit even when not significant
- numbers are rounded to the nearest last zero digit after the decimal separator
  - `format-number( .204, ".00" )` returns `".20"`
  - `format-number( .205, ".00" )` returns `".21"`
- `format-number( .9876, "0.0%" )` returns `"98.8%"`
  - percent triggers automatic multiplication by 100 and display of suffix
- `format-number( .9876, "0.0&#x2030;" )` returns `"987.6‰"`
  - per-mille triggers automatic multiplication by 1000 and display of suffix
- `format-number( -12345, "#,###;(#)" )` returns `"(12,345)"` *not* `"(12345)"`
  - the semi-colon separates the positive value pattern from the negative
  - only the prefix and suffix of the negative pattern are respected, the whole and fraction parts are ignored
- `format-number( 1234.56, "# ###,00&#x20ac;", 'myeuro' )` returns `"1 234,56€"`
  - uses cultural practices described in a declared decimal format named `'myeuro'`
- the format string syntax is the same as that used in Java and is described in more detail in the next section regarding Decimal Formatting
- `format-number()` is specific to XSLT and is not part of XPath

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Decimal formatting in XSLT (cont.)

`format-number()` third argument:

- name of a declaration of cultural influences on the formatting of numbers
- omitted to specify the default or unnamed decimal format

`<xsl:decimal-format` *`attributes`*`/>`

- top-level instruction describing grammar components and formatting strings
- `name="`*`description-qname`*`"`
  - cannot be more than one unnamed specification
  - cannot be more than one named specification with the same name
  - exception accommodating external stylesheet fragments:
    - two format declarations with the same name (or unnamed) must have identical effective values
- `pattern-separator=` between positive and negative formats (`';'`)
- `digit=` for insignificant digits (`'#'`)
- `zero-digit=` for significant digits (`'0'`)
- `decimal-separator=` between whole and fraction (`'.'`)
- `grouping-separator=` between number groups (`','`)
- `percent=` for calculation and display (`'%'`)
- `per-mille=` for calculation and display (`'&#x2030;'`)
- `minus-sign=` for presentation of negative numbers (`'-'`)
- `infinity=` for presentation of infinity values (`"Infinity"`)
- `NaN=` for presentation of invalid numbers (`"NaN"`)

Example:

```
01 <xsl:decimal-format name="myeuro" NaN="(invalid)"
02                     decimal-separator="," grouping-separator=" "/>
```

T1 `java.text.DecimalFormatSymbols.html`

- each get/set method pair in the documentation is an attribute defined for the `<xsl:decimal-format>` element
- the currency symbol (0x00a4) cannot be used in a format pattern for portability reasons because this was added after the initial release of JDK 1.1
- the quote character is not localized

T1 The formal specification of the format string syntax is that used for Java:
`http://java.sun.com/products/jdk/1.1/docs/api/java.text.DecimalFormat.html`

P2 The formal specification of the format string syntax is in the XSLT 2.0 specification

- `http://www.w3.org/TR/xslt20/#processing-picture-string`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Decimal formatting in XSLT (cont.)

`format-number()` second argument:

- a user-specified sequence of formatting characters interpreted by the processor when formatting the numeric value of the first argument to the function
- one or two format sub-sequences
  - the first sub-sequence for positive value and the optional second for negative
    `{p-prefix}p-whole{p-fraction}{p-suffix}`
    `{pat-sep{n-prefix}n-whole-ignored{n-fraction-ignored}{n-suffix}}`
  - each sequence is made of up to four components (braces indicate optional components):
    - a prefix that is output verbatim
      - provided there are no digit or zero-digit characters
    - the whole number component
      - comprised of optional excess digit indicators, followed by leading zero-digit indicators, interspersed with grouping separators, optionally followed by a decimal separator
      - ignored for negative numbers
        - sufficient to utilize as a minimum either a single digit indicator or just the decimal separator, but not nothing
        - whole number component for positive number is utilized for both
    - a fraction number component
      - comprised of trailing zero-digit indicators that force trailing zeroes, followed by optional excess digit indicators where trailing zero-digits are suppressed
      - ignored for negative numbers
        - fraction number component for positive number is utilized for both
    - a suffix that is output verbatim and checked for a percent or per-mille indication
      - provided there are no digit or zero-digit characters
  - separated by the pattern separator character when both are specified
    - the negative number pattern is not required if the prefix and the suffix for the negative number are formatted the same as for the positive number
  - characters not defined by the grammar are simply copied

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Decimal formatting in XSLT (cont.)

The following lines are extracted from `dformat.xsl` to illustrate examples of using format patterns:

```
<xsl:text>1 + '$###,####,##.###0;(########.000#)' </xsl:text>
<xsl:value-of select="format-number(123456789.9876,
                          '$###,####,##.###0;(########.000#)')"/>
<xsl:text>&nl;2 - '$######,###.###;($######,##.####)' </xsl:text>
<xsl:value-of select="format-number(-123456789.9876,
                          '$######,###.###;($######,##.####)')"/>
<xsl:text>&nl;3 + '+#########.##;(##,##,##,##.000#)'  </xsl:text>
<xsl:value-of select="format-number(123456789.9876,
                          '+#########.##;(##,##,##,##.000#)')"/>
<xsl:text>&nl;4 - '$######,###0.#;$-#'                </xsl:text>
<xsl:value-of select="format-number(-123456789.9876,
                                    '$######,###0.#;$-#')"/>
<xsl:text>&nl;5 + '$######,#00.####'                  </xsl:text>
<xsl:value-of select="format-number(123456789.9876,
                                      '$######,#00.####')"/>
<xsl:text>&nl;6 - '$#######,##0.000#;(-$#)'           </xsl:text>
<xsl:value-of select="format-number(-123456789.9876,
                                     '$#######,##0.000#;(-$#)')"/>
<xsl:text>&nl;7 + '###,###.###%;(#)'                  </xsl:text>
<xsl:value-of select="format-number(123456789.9876,
                                       '###,###.###%;(#)')"/>
<xsl:text>&nl;8 - '###,###.###%;(#)'                  </xsl:text>
<xsl:value-of select="format-number(-123456789.9876,
                                       '###,###.###%;(#)')"/>
<xsl:text>&nl;9 + '#######,###.###%'                  </xsl:text>
<xsl:value-of select="format-number(123456789.9876,
                                       '#######,###.###%')"/>
<xsl:text>&nl;10 - '#######,###.###%'                 </xsl:text>
<xsl:value-of select="format-number(-123456789.9876,
                                       '#######,###.###%')"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Decimal formatting in XSLT (cont.)

The following lines are produced when running the stylesheet `dformat.xsl`:

```
01 1 + '$###,####,##.###0;(########.000#)' $1,23,45,67,89.9876
02 2 - '$######,###.###;($######,##.####)' ($123,456,789.988)
03 3 + '+#########.##;(##,##,##,##.000#)'  +123456789.99
04 4 - '$######,###0.#;$-#'                $-1,2345,6790
05 5 + '$######,#00.####'                  $123,456,789.9876
06 6 - '$#######,##0.000#;(-$#)'           (-$123,456,789.9876)
07 7 + '###,###.###%;(#)'                  12,345,678,998.76%
08 8 - '###,###.###%;(#)'                  (12,345,678,998.76)
09 9 + '#######,###.###%'                  12,345,678,998.76%
10 10 - '#######,###.###%'                 -12,345,678,998.76%
```

Note the following regarding each numbered line above when using a Java-based implementation:

1 the number of digit indicators between the decimal separator and the closest grouping separator dictate the groupings for the entire whole number component (other grouping sizes are ignored)
2 the grouping indications in the negative value pattern are ignored; the whole number and fraction component formatting is dictated by the positive value pattern even when the value is negative
3 the second digit of the fraction component is the rounded value of the third digit ("8" becomes "9" because of "7")
4 a single digit indicator suffices to delimit the prefix from the suffix in the negative value pattern
5 any number of zero digits can lead the decimal separator
6 any number of zero digits can follow the decimal separator
7 the use of the percent indicator in the suffix triggers the value to be multiplied by 100 before being formatted
8 illustrating the problem with reliance on the incorrect Java-library, the use of the percent indicator in the positive value triggers the multiplication for the negative value even though the negative value doesn't include the percent indicator; the result should be "-123456789.9876"
9 a complete specification can omit the pattern for negative values
10 a specification omitting the pattern for negative values defaults to the value being prefixed by the minus sign

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Decimal formatting in XSLT (cont.)

The attributes of the named decimal format declared by an `<xsl:decimal-format>` instruction specify the characters that control the interpretation of a format pattern and/or the formatting of a numeric value:

- attributes only controlling the interpretation of characters in the grammar of a format pattern (never in the result of formatting a number value) are as follows:
  - `pattern-separator=` (e.g. ";")
  - `digit=` (e.g. "#")
  - `zero-digit=` (e.g. "0")
- attributes both controlling the interpretation of characters in the grammar of a format pattern and specifying the characters that may appear in the resulting string when the function is formatting a number are as follows:
  - `decimal-separator=` (e.g. "." (default) or ",")
  - `grouping-separator=` (e.g. "," (default) or a space)
  - `percent=` (e.g. "%")
  - `per-mille=` (e.g. Unicode character 0x2030)
- attributes only specifying the characters and strings that may appear in the result of formatting a number value are as follows:
  - `minus-sign=` (e.g. "-" (hyphen/minus, 0x2d))
  - `infinity=` (e.g. Unicode character 0x221e; the default value is the string "Infinity")
  - `NaN=` "Not-a-Number" (e.g. Unicode character 0xfffd; the default value is the string "NaN")

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

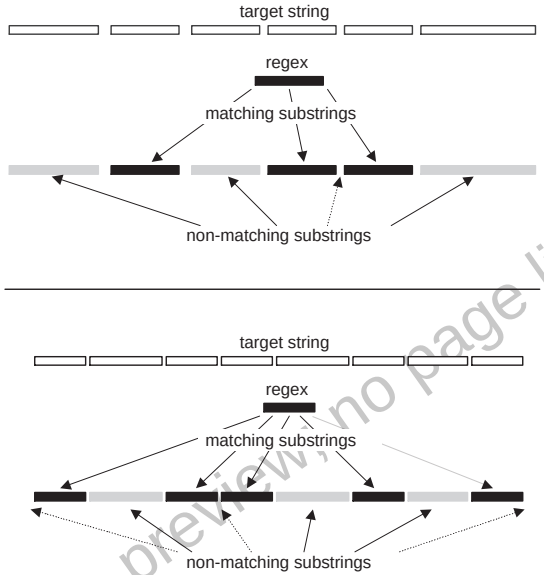
Regular expressions

Regular expressions are very powerful "wild-card" patterns of letters and symbols

- used to detect the presence of actual characters found in strings
- used to parcel portions of a string in order to act on the parcels
- a regular expression describes a pattern of characters searched for and
- numerous online tutorials for the writing of regular expressions

Strings are comprised of "matching" (the regular expression) and "non-matching" substrings

- the regular expression defines the pattern of a matching substring
  - substrings that do not match the regular expression are as long as possible
- there are never two consecutive matching substrings in sequence, nor two consecutive non-matching substrings in sequence
  - there are empty non-matching substrings between two matching substrings
- there is always a non-matching substring at the start and end of the set of substrings
  - when the string begins with a matching substring there is an empty non-matching substring between the start of the string and the first matching substring
  - when the string begins with a matching substring there is an empty non-matching substring between the start of the string and the first matching substring



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Regular expressions (cont.)

Examples of matching against "`pqrabbbcstu`":

1. "`abc`" doesn't match anywhere
   - there are too many "`b`"s
2. "`ab+c`" matches "`abbbc`"
   - the "`+`" allows one or more "`b`"s
3. "`ab*c`" matches "`abbbc`"
   - the "`*`" allows zero or more "`b`"s
4. "`ab+x*c`" matches "`abbbc`"
   - the "`*`" allows zero or more "`x`"s, and there are none
5. "`^ab+c`" doesn't match
   - the "`^`" represents the start of the string
6. "`r[bca]+s?x?t`" matches "`rabbbcst`"
   - the "`[bca]`" is a character class choice of "`b`", "`c`", or "`a`", and "`s`" may or may not be there, and "`x`" may or may not be there
7. "`[m-w]+`" matches "`pqr`" and "`stu`"
   - this use of "`-`" in a character class is a range indication

Examples of matching against "`a bcb aa bcb a`":

1. `(..)aa.*\1` matches "`b aa bcb `"
   - the parenthesis group any two characters in advance of two "`a`"s, followed by any sequence up to the same two characters that were matched in the group
2. `a.*a` matches the entire string
   - the longest possible string of any characters between two letters "`a`"
3. `a.*?a` matches "`a bcb a`" twice
   - the shortest possible string of any characters between two letters "`a`"
4. `\s+` matches four times
   - sequences of white-space characters
5. `\S+` matches five times
   - sequences of non-white-space characters

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Regular expressions (cont.)

- *regex* represents a subset of Perl and Unix regular expressions
  - full specification in section 7.6.1 of "Functions and Operators"
    - http://www.w3.org/TR/xpath-functions/#regex-syntax
    - extends the specification of Annex F of W3C Schema data types
      - http://www.w3.org/TR/xmlschema-2/#regexs
  - the spec cites http://www.unicode.org/unicode/reports/tr18/ for Unicode
    - http://unicode.org/Public/UNIDATA/Blocks.txt is useful
  - whereas W3C Schema regex is anchored, XPath 2.0 regex is unanchored
- Some helpful regular expression patterns (not nearly a complete list):
  - "\t", "\r", "\n", match tab, return and newline
  - "." matches any character except \n and \r
  - "^" and "$" match the start and end of the string
  - "(" and ")" wrap groups (numbered by order of "(" characters)
  - "*digit* matches previously numbered wrapped group
  - "*regex*|*regex*" matches one of two (or more) specified expressions
  - quantifiers of longest matching expression (or shortest if a "?" follows):
    - "*regex*?" matches zero or one atomic expressions
    - "*regex**" matches zero or more atomic expressions
    - "*regex*+" matches one or more atomic expressions
    - "*regex*{n}" matches *n* atomic expressions
    - "*regex*{n,}" matches *n* or more atomic expressions
    - "*regex*{n,m}" matches *n* through *m* atomic expressions
  - "\s", "\i", "\c", "\d", "\w", "\p" matches (respectively) one white-space, name-start, name, digit, word and Unicode property character
    - using upper case implies "not" of the characters
  - "*char*" matches the pattern character instead of interpreting it
  - "[*chars*]" matches class of any of the individual or range of characters listed
    - a range is any "*low*-*high*" as in "a-z" or "A-Z"
    - an exclusion is any "*property-class*-*class*" as in "[\c-[:]]"
  - "[^*chars*]" matches any of other than the individual or range of characters listed
- *flags* impacts the interpretation of the regular expression
  - s - includes \n in the "." wild-card
  - m - treats "^" and "$" on a per-line basis (according to \n)
    - instead of on the basis of the entire string
  - i - treats the characters without consideration of letter-case
  - x - collapses any white-space characters in the pattern prior to interpretation
    - does not remove white-space found in character classes (using "[" and "]")
    - allows breaks and indentation in the expression for legibility

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Regular expressions (cont.)

Recall the definitions of matching and non-matching substrings on page 300

- matches(*string*,*regex-match*,*optional-flags*)
  - returns boolean true if there exists any matching second argument regular expression substring anywhere in the first argument string, or false if are there are none
- tokenize(*string*,*regex-match*,*optional-flags*)
  - returns the sequence of all non-matching substrings found in the first argument as defined by the regular expression pattern second argument
    - occurrences of the second argument are not included in the sequence
    - the second argument cannot evaluate to the empty string

e.g. tokenize('a bcb aa bcb a','\s+') returns "('a','bcb','aa','bcb','a')"

- split a string at sequences of white-space characters (not keeping the white-space)

e.g. tokenize(unparsed-text('file.txt'),'\r?\n|\r')[normalize-space(.)]

- read a text file, splitting the file into lines at all sequences of an optional carriage return followed by a linefeed, returning only those lines that are not only white-space characters

Beware unexpected results for tokenize(" a ","\s+")

- returns three items ("", "a", "")
- using normalize-space() cleans up a string before tokenizing with "\s+"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Regular expressions (cont.)

Recall the definitions of matching and non-matching substrings on page 300

F2 `replace(`*`string`*`,`*`regex-match`*`,`*`regex-replace`*`,`*`optional-flags`*`)`

- returns the first argument with each matching second argument regular expression pattern replaced with the third argument
- "$*n*" in the replacement string uses the matched group
  - counting from 1 by the "`(`" characters used to capture groups
  - the group `$0` represents the entire matched sequence

e.g. `replace('abcxacyabbbc','a.*?c','z')` returns "`zxzyz`"

- without the "`?`", the entire string would have been replaced with a single "`z`"

e.g. `replace('abcxacyabbbc','([^abc]+)','$1$1')` returns "`abcxxacyyabbbc`"

- any sequence of characters other than "`a`", "`b`" or "`c`" is duplicated in the return string

e.g. `replace('abcxacyabbbc','([ap])(.*?)([qc])','$3$2$1')` returns "`cbaxcaycbbba`"

- reverse the prefix and suffix of the shortest sequence that satisfies "`a`" or "`p`" as the prefix and "`q`" or "`c`" as the suffix

e.g. `replace(.,'([\[(.:/{\\])','$1')` injects invisible breaking characters into string

- replace punctuation characters "`[`", "`(`", "`.`", "`:`", "`/`", "`{`" and "`\`" with itself followed by a Unicode zero-width space
- useful for massaging a URL in such a way that forced line breaks are triggered at controlled locations in the string

e.g. `replace(@name, ".*", " name=""$0""")` returns a specification string

- equivalent to `concat(" name=""",@name,"""")` when the attribute is present

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

String analysis in XSLT 2.0

Recall the definitions of matching and non-matching substrings on page 300

I2 Exploiting regular expressions in the analysis of strings

```
<xsl:analyze-string select="string-value" regex="AVT" flags="AVT">
  optional <xsl:matching-substring>
  optional <xsl:non-matching-substring>
</xsl:analyze-string>
```

- wrap at most one `<matching-substring>` followed by at most one `<non-matching-substring>` instructions governing the disposition of, respectively, matched and unmatched components of the given `select=` expression
  - the order of the child instructions is important
  - at least one of the child instructions must be present
- the `regex=` is an attribute value template describing what constitutes a matching string
  - note that if you want brace brackets in the expression they have to be doubled or they would be interpreted as an `<xsl:value-of>`
- the `flags=` attribute has the same interpretation as the "*flags*" parameter of the `matches()` call
- the child instructions are invoked in the order that matching and non-matching substrings of the analyzed string are encountered

```
<xsl:matching-substring>
  ...template...
</xsl:matching-substring>
```

- add to the result tree or temporary tree the specified template for each of the matching substrings of the analyzed string

```
<xsl:non-matching-substring>
  ...template...
</xsl:non-matching-substring>
```

- add to the result tree or temporary tree the specified template for each of the non-empty non-matching substrings of the analyzed string
  - note that this means empty non-matching substrings are ignored and never processed in any way

`.`, `position()` and `last()` reflect the set of substrings

- not including empty non-matching substrings

Captured nested substrings can be accessed in `<xsl:matching-substring>`

- `regex-group(`*`integer`*`)`
- integer = 0 returns the entire matching string
- integer = n returns the nth captured sub-expression
  - a regular-expression sub-expression begins with a left parenthesis "`(`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

String analysis in XSLT 2.0 (cont.)

Recall the definitions of matching and non-matching substrings on page 300

Example analyses of the string "abcxacyabbbc":

```
<xsl:analyze-string select="'abcxacyabbbc'" regex="a.*?c">
  <xsl:matching-substring>
    <xsl:value-of select="concat('{',.,'}')"/>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
    <xsl:value-of select="concat('(',.,')')"/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
```

Constructs the string: "{abc}(x){ac}(y){abbbc}"

- each matching substring is surrounded by "{" and "}"
- each non-empty non-matching substring is surrounded by "(" and ")"
- all empty non-matching substrings are ignored

```
<xsl:analyze-string select="'abcxacyabbbc'" regex="a.*?c">
  <xsl:matching-substring>
    <b>
       <xsl:value-of select="."/>
    </b>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
    <xsl:value-of select="."/>
  </xsl:non-matching-substring>
</xsl:analyze-string>
```

Constructs the marked-up result: "abcxacyabbbc"

- each matching substring is wrapped in an element named "b"
- each non-empty non-matching substring is output as is
- all empty non-matching substrings are ignored

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using node-set-related expression functions

Node-set operators and functions:

- node-sets obtained from source trees
- effective Boolean value of an empty node set is `false`, all other node sets test `true`

operator `|`

- union of node-sets into a new node-set
- the same node is never duplicated in a node set
- result set is always ordered in document order

P2 operator `union`

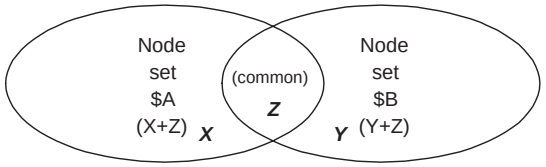
- return the union of members of both operands while eliminating duplicates

P2 operator `intersect`

- return the members that are found in both operands while eliminating duplicates

P2 operator `except`

- return the members of the left operand that are not found in the right operand



`"$A union $B"` returns ***X*** + ***Y*** + ***Z*** (no duplicates)
`"$A intersect $B"` returns ***Z*** (no duplicates)
`"$A except $B"` returns ***X***

The one expression used to get only both portions named "X" and "Y" found in the diagram:

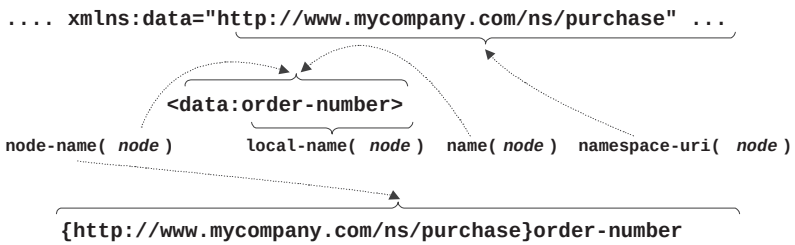
- `"($A union $B) except ($A intersect $B)"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using node-set-related expression functions (cont.)

The following illustrates name-related functions that all optionally use a node-set argument:

- operate on the current node if no argument supplied
- operate on the give node if a singleton node set supplied
- P1 operate on the first node found in a node set argument of more than one node
- P2 trigger a runtime error for a node set argument of more than one node
- return the empty string for an empty node set argument



P1 `namespace-uri(`*`optional-node-set`*`)`

`namespace-uri(`*`optional-node`*`)`

- returns the namespace name (URI) of the qualified namespace name as a string

P1 `local-name(`*`optional-node-set`*`)`

`local-name(`*`optional-node`*`)`

- returns the local part of the node name as a string
- does not include the namespace prefix if one was used for the node

P1 `name(`*`optional-node-set`*`)`

`name(`*`optional-node`*`)`

- `name()` returns the qualified name as a string
- includes the namespace prefix if one was used for the node

P2 `node-name(`*`optional-node`*`)`

- `node-name()` returns the expanded name as a QName comparison value
- the lexical value is the node's name including any present prefix

Recall the discussion regarding namespace terminology (page 30).

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Calculating values using node-set-related expression functions (cont.)

Chapter 7 - Data type expressions and functions
Section 4 - Node-set expressions

P2 `nilled(`*`optional-node`*`)`

- returns Boolean indication of the node being an element that has been nilled according to W3C Schema definition

P2 `data(`*`optional-item-sequence`*`)`

- returns the typed values of members of the sequence as a sequence
- recall the discussion of data model node values on page 77

P2 `base-uri(`*`optional-node`*`)`

- returns the absolute base URI for the given node or the current node if no node is specified
  - accommodates the presence of `xml:base` in the tree
- useful for determining absolute URI values from relative URI strings
- recall the physical hierarchy illustrated on slide 9
  - the base URI of a node found in the fragment from "`d.xml`" is the absolute URI of the "`ddir/d.xml`" as shown at the left
  - when used for URI resolution only the absolute directory up to "`ddir`" is used

P2 `static-base-uri()`

- returns the absolute base URI string for the instruction node in the operation tree

P2 `root(`*`optional-node`*`)`

- returns the root node of the node tree for the given node or the current node if no node is specified
  - irrespective of the entity structure, the root node is the top of the entire XML document

P2 `document-uri(`*`optional-root-node`*`)`

- returns the absolute document URI for the given node, or the current node if no node is specified, only if the supplied node is a document node
  - otherwise the empty sequence is returned
- it is possible that the tree of nodes was not created from a URI, thus the function will return the empty string
  - e.g. a temporary tree
  - e.g. an in-memory-only source tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Node-set intersection and difference in XSLT 1

Sometimes it is necessary to determine the intersection or difference of two node-sets selected from the source tree. The following expression, colloquially referred to as the "Kaysian Method" after Mike Kay who first proposed this use of XPath, will select only those nodes that are in both of two node-set variables named `set1` and `set2`:

```
- $set1[count(.|$set2)=count($set2)]
```

The above expression takes advantage of the XPath union operator to determine that a given node doesn't impact on the count of nodes of the union of the node and the second node-set. The predicate returns true when the count isn't affected, thus including the member being tested. It is not necessary to test the members of the second set because the intersection would have to include members of the first set, all of which are being tested in the above expression.

The symmetric difference requires an expression involving the union of the determination of those nodes in each set that are not in the other set:

```
- (   $set1[count(.|$set2)!=count($set2)]
   | $set2[count(.|$set1)!=count($set1)] )
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Node-set intersection and difference in XSLT 1 (cont.)

The following script illustrates the assignment of two node-set variables from a common area of the source tree (in this case the stylesheet is also the source tree) and the intersection and symmetric difference of those two variables:

```
<?xml version="1.0"?><!--intrdiff.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:data="crane"
                version="1.0">

<xsl:output method="text"/>

<data:data>              <!--data source for testing purposes-->
  <item>1</item><item>2</item><item>3</item>
  <item>4</item><item>5</item><item>6</item>
</data:data>

<xsl:template match="/">                          <!--root rule-->
  <xsl:variable name="ns1" select="//item[position()>1]"/>
  <xsl:variable name="ns2" select="//item[position()&lt;5]"/>

  <xsl:for-each select="$ns1[count(.|$ns2)=count($ns2)]">
    Intersection: <xsl:value-of select="."/>
  </xsl:for-each>
  <xsl:for-each select="(   $ns1[count(.|$ns2)!=count($ns2)]
                          | $ns2[count(.|$ns1)!=count($ns1)] )">
    Difference: <xsl:value-of select="."/>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

When run with itself as input, the following is the result:

```
    Intersection: 2
    Intersection: 3
    Intersection: 4
    Difference: 1
    Difference: 5
    Difference: 6
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

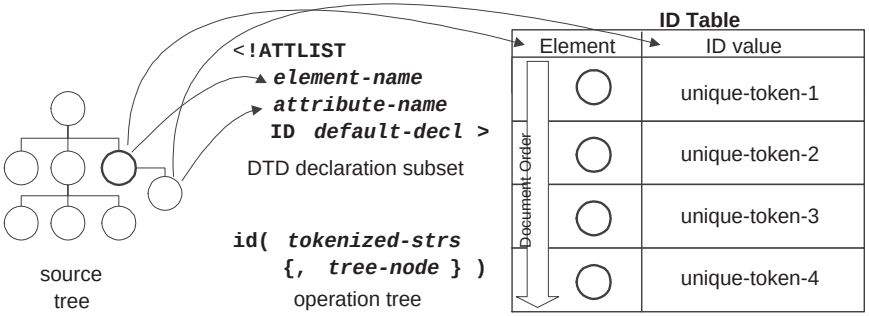
User XML identifier referencing

Built-in table of elements keyed by their user-specified unique identifier:

Recall that element nodes in the source node tree can have unique identifiers defined by the presence of XML attributes of type `ID`. It is a common practice to implement cross referencing from other elements using XML attributes of type `IDREF` or `IDREFS` that point to the unique identifiers of elements with the corresponding `ID` attribute.

- a validity constraint that `IDREF` be an XML name
- a validity constraint that `IDREFS` be one or more XML names

The following illustrates how a processor maintains `ID` information from an XML instance:



P1 Requires the presence of DTD declarations

- `ATTLIST` declaration indicates the "`ID`-ness" of an attribute
- the name of the attribute is irrelevant

P2 Requires the presence of either DTD declarations or schema-awareness of W3C Schema declarations

- an attribute is not known to be of type `ID` unless it is declared such

P2 Reserved `xml:id=` attributes are automatically recognized

- no declarations are required as the semantic of the reserved name is recognized

Recall Attribute node (page 84)

- one can declare a minimal DTD containing relevant ID declarations

The table does not include generated identifiers from the node tree

- recall from page 77 that every node of every tree has an automatically-generated identifier created anew with every transform using a custom alphanumeric pattern chosen by the processor

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

User XML identifier referencing (cont.)

Finding elements by their unique identifier:

`id(`*`non-node-set-value-converted-to-string`*`)`

P2 `id(`*`non-node-set-value-converted-to-string`*`,`*`tree-node`*`)`

- returns the union of all nodes with identifier equal to *tokenized* string
- converts the argument into a string as with `normalize-space()`
- splits the resulting string into set of tokens
- find element nodes whose unique identifier is in the set of tokens
- the tree node specifies in which tree the search is done and if not specified the search is in the same document as the current node

`id(`*`node-set`*`)`

P2 `id(`*`node-set`*`,`*`tree-node`*`)`

- returns the union of the call to `id()` with the string value of each of the nodes
- note that the nodes used in the node-set argument need not be attribute nodes, and if any one is an attribute node, it need not be an attribute of XML type `IDREF`, as the value obtained is just used as a string
- of course the node could, indeed, be an attribute node of type `IDREF`

See Document referencing in XSLT (page 268) for an important consideration regarding addressing unique identifiers in multiple source documents.

A separate `ID` table is maintained for each source document

- the table that is accessed is that for the document in which the current node is found

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

User XML identifier referencing (cont.)

Examples of being used stand-alone:

- `id("buyer")`
  - returns the element whose unique identifier is the string value "`buyer`"
- `id(" buyer seller ")`
  - returns the elements whose unique identifier is either the string value "`buyer`" or "`seller`"
- `id(@where)`
  - returns the element whose unique identifier is the same value as the value of any of the space-separated tokens in the `where=` attribute of the current node
- `id(where)`
  - returns the element whose unique identifier is the same value as the value of any of the space-separated tokens in all of the `where` children elements of the current node
    - note in this example that one is not obliged to use `IDREF` attributes for the argument to the function
    - any string can be used as an argument

Can be first step in a multiple-step location paths and can have predicates applied to the returned node-set:

- `id(@where)/phone[2]`
  - returns the set of the second `phone` child elements of all elements referenced by the identifiers found in the `where=` attribute of the current element node
- P2 `id(@where,$tree)/phone[2]`
  - here the variable `$tree` is a singleton node set with a node from the desired tree
- P2 `$trees/id($where,.)/phone[2]`
  - here the variable `$trees` is an arbitrary node set from which one node at a time is passed to the `id()` function
  - `$where` is a variable with the tokens to be searched in all trees
  - the return set is the union of all second phone number children of those nodes referenced by the tokens in the `$where` variable that are found in all trees

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

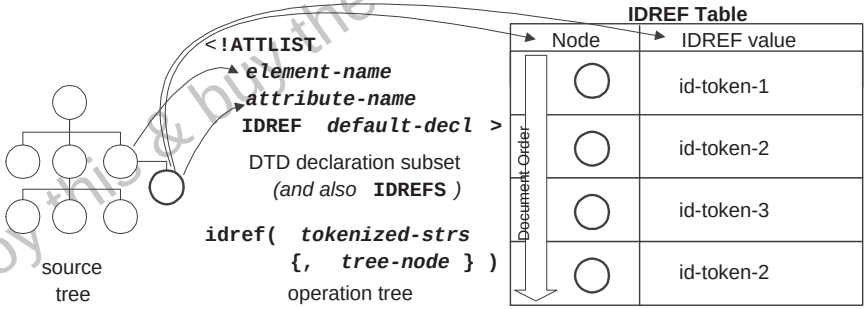
User XML identifier referencing (cont.)

P2 Finding pointers to unique identifiers:

Elements with attributes of type `ID` can be referred to by attributes of type `IDREF` or `IDREFS`

- W3C Schema allows elements to also be declared as type `IDREF` or `IDREFS`
- many nodes can be pointers to a single element with an `ID`-typed attribute

P2 The following illustrates how a processor maintains `IDREF` information from an XML instance:



P2 Requires the presence of either DTD declarations or schema-awareness of W3C Schema declarations

- an attribute is not known to be of type `ID` unless it is declared such

P2 `idref(`*`non-node-set-value-converted-to-string`*`)`

P2 `idref(`*`non-node-set-value-converted-to-string`*`,`*`tree-node`*`)`

P2 `idref(`*`node-set`*`)`

P2 `idref(`*`node-set`*`,`*`tree-node`*`)`

- returns all of the element or attribute nodes declared as type `IDREF` that are referencing the supplied set of values interpreted as `ID` values

P2 A separate `IDREF` table is maintained for each source document

- the table that is accessed is that for the document in which the current node is found

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data-model identifier referencing

Chapter 7 - Data type expressions and functions
Section 4 - Node-set expressions

Authored identifiers are only unique within the scope of a single XML document

- may be duplicated in an aggregation when combining multiple XML documents
  - at which point they are no longer unique
- recall that every node has a unique system-generated identifier (page 77)

1.1 `generate-id(`*`optional-node-set`*`)`

1.2 `generate-id(`*`optional-node`*`)`

- returns an opaque implementation-dependent string that should be used blindly as a unique identifier for the node as a string
  - lexically parsed as a valid XML name with only alphanumeric characters (no "_" or ".")
    - not guaranteed to be exclusive of ID/IDREF values
  - operate on the current node when no argument is supplied
  - 1.1 operate on the first node found in a non-empty node-set argument
  - 1.2 an error is reported if more than one node is supplied
  - returns an empty string for an empty node-set argument
- the string is different for every node across all node trees
- the string is persistent for each node through only the given invocation of the XSLT processor and a given value cannot be relied upon from any other invocation
- very important when aggregating from multiple XML source trees
  - authored ID values in multiple trees are not mutually exclusive
  - generated ID values in multiple trees are guaranteed to be mutually exclusive
- when visiting the node at the time of anchoring the reference:
  - HTML: `<a name="{generate-id(.)}">`
  - XML: `<`*`element`* `id="{generate-id(.)}">`
- can be used to synthesize hyperlinks where ID/IDREF is available
  - when visiting the referring node:
    - HTML: `<a href="#{generate-id(id(@idref))}">`
    - XML: `<`*`element`* `idref="{generate-id(id(@idref))}">`
- can be used to synthesize hyperlinks where ID/IDREF unavailable
  - when visiting the node at the time of making the reference:
    - HTML: `<a href="#{generate-id(.)}">`
    - XML: `<`*`element`* `idref="{generate-id(.)}">`
- can be used to compare two nodes as being the same node
  - simple equality is insufficient because it uses the value of a node
  - using the generated identifier is sufficient because of the uniqueness constraint
  -
```
test="generate-id($a)=
      generate-id($b)"
```
  - 2.2 `test="$a is $b"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data-model identifier referencing (cont.)

Chapter 7 - Data type expressions and functions
Section 4 - Node-set expressions

Examples of using `generate-id()`:

Identification of node during processing:

- consider when the node's position has to be identified in a different context than the current node list in which the node is being processed
- since the node's generated identifier is persistent for the execution of the script (but not after the script has completed), it can be assigned to a variable and checked at any time
- in this example, the template is being called from elsewhere in the script and the location of the current node is evaluated in an arbitrary node list (not the current node list when the template was invoked):

```
01 <xsl:template name="node-position-in-node-and-attr-set">
02   <xsl:variable name="this-id" select="generate-id(.)"/>
03      <!--current node list of parent's children and attributes-->
04   <xsl:for-each select="../node()|../@*">
05     <xsl:if test="generate-id(.)=$this-id"> <!--found saved id-->
06       <xsl:value-of select="position()"/><!--using new context-->
07     </xsl:if>
08       <!--note the performance hit by having to go through all
09           the nodes even *after* finding the desired node-->
10   </xsl:for-each>
11 </xsl:template>
```

Unique naming of nodes for output stream:

- consider the need to generate a hyperlinked table of contents where there is no handy XML `ID` construct to be used to uniquely identify the elements:

```
01 <?xml version="1.0"?>
02 <test>
03 <section><title>First Section</title>
04 <para>The content of the section is here.</para></section>
05 <section><title>Second Section</title>
06 <para>The content of the section is here.</para></section>
07 <section><title>Third Section</title>
08 <para>The content of the section is here.</para></section>
09 </test>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Data-model identifier referencing (cont.)

The following script will use the XSLT-processor-generated node identifiers in place of XML `ID` identifiers:

```
<?xml version="1.0"?><!--genid.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">
  <html>
    <h2>Table of Contents</h2>
    <xsl:for-each select="//section/title">
      <a href="#{generate-id(.)}"><xsl:value-of select="."/>
      </a><br/>
    </xsl:for-each>
    <xsl:apply-templates/>
  </html>
</xsl:template>

<xsl:template match="title">
  <h2><a name="{generate-id(.)}"><xsl:value-of select="."/>
      </a></h2>
</xsl:template>

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

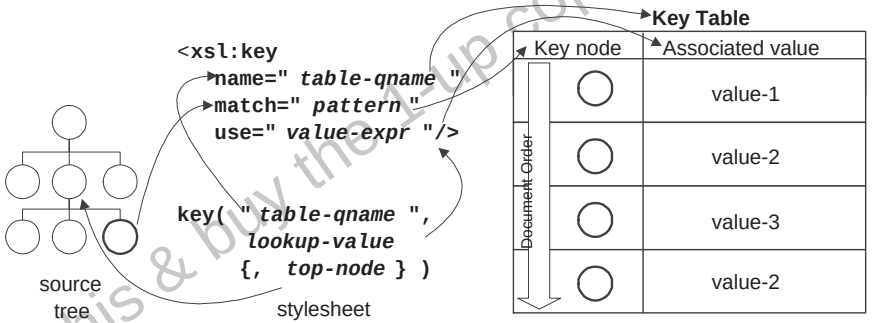
</xsl:stylesheet>
```

Note again that the persistence of the values is guaranteed only within the execution of the script and not afterwards to any subsequent execution of the script. The above technique is not appropriate when needing to point into the result from other files in a persistent manner (where using `ID` would be useful for that purpose).

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT key node referencing

The stylesheet can identify key nodes of the source tree for subsequent fast access by the XSLT processor:



1 `<xsl:key name="table-qname" match="pattern" use="string-expr"/>`

2 `<xsl:key name="table-qname" match="pattern" use="value-expr"/>`

2 `<xsl:key name="table-qname" match="pattern" use="value-expr" collation="uri"/>`

- identifies all source tree nodes for each declared key for stylesheet manipulation with a simple name
- builds lookup table of member items based on an equality test for the lookup value
- e.g.: relate all employee records to the employee's respective manager's employee record matching each employee's `ManagedBy` child element's value with the corresponding employee record with the same value for its `emp=` attribute

The XSLT processor can optimize searching in the lookup table

- all values are fixed after the source tree is processed
- the processor can index the table for quick retrieval based on lookup value
- faster return of selected nodes than traversing the axes using XPath expressions and predicates evaluated at the point of reference

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT key node referencing (cont.)

Establishing a lookup table of key relationships in a declarative fashion

When indexing behavior similar to XML `ID`/`IDREF` attribute cross-referencing values is required by the content of either an attribute or an element, the stylesheet writer can define and reference key values reflecting the implicit cross-references:

- keys have three aspects of definition in the `<xsl:key>` top-level instruction:
  - the namespace-qualified name of the key
    - `name="`*`qname-of-key-table`*`"`
    - the collection of key nodes distinguished from other collections of key nodes
    - multiple declarations with the same name cumulatively build the members in the collection of that name
  - the nodes that have the key
    - `match="`*`identifying-pattern-of-nodes-in-key-table`*`"`
    - a XPath matching pattern expression
      - similar to `count=` attribute in `<xsl:number/>`
  - the lookup values of the keyed nodes
    - `use="`*`expression-evaluating-the-value-of-key-in-the-set`*`"`
    - an XPath selection expression
      - if relative, then evaluated relative to each keyed node
    - expression evaluated to a string
    - string values need not be unique
      - nodes with like values are ordered in document order
  - the lookup values of the keyed nodes
    - 2 `collation="`*`collation-overriding-default-collation`*`"`
- the index built for a key is somewhat dissimilar to `ID`/`IDREF`
  - key values in the set need not be unique
    - there can be multiple key nodes in a document with the same node, same key name, but different key values
    - there can be multiple key nodes in a document with the same key name, same key value, but different nodes
  - a key's value need not be parsed as an XML name token
    - a node relative to the key node can represent the key's value
    - a key's value can be the result of an arbitrary expression evaluated with the key node as the current node
  - a lookup value need not be present in the collection

Note that a variable reference cannot be used in either the `match=` or `use=` expressions.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT key node referencing (cont.)

Using a key

1 `key(`*`table-qname`*`,`*`non-node-set-value-converted-to-string`*`)`

1 `key(`*`table-qname`*`,`*`non-node-set-value-expression`*`)`

2 `key(`*`table-qname`*`,`*`non-node-set-value-expression`*`,`*`top-node`*`)`

- returns that subset of key nodes whose indexed key value is equal to the given string lookup value
- if the top node is specified then all nodes returned are those found at or below the given node

`key(`*`table-qname`*`,`*`node-set`*`)`

2 `key(`*`table-qname`*`,`*`node-set`*`,`*`top-node`*`)`

- returns the union of the call to `key()` with the value of each of the nodes in the node set as a lookup value
- if the top node is specified then all nodes returned are those found at or below the given node

Can have predicates applied to the returned node-set:

- `key('taxes', 'Canada')[1]`
  - returns only the first node of all nodes in the '`taxes`' key table whose lookup value is the string '`Canada`'
  - like lookup values are returned in document order of the nodes indexed, thus, this returns the first such node in document order
- `key('taxes', 'Canada')[@type='federal'][1]`
  - of all nodes returned by the key function, return only the first whose `type=` attribute is '`federal`'

2 The third argument must be a singleton node

- consider an example where the variable `$seq` has more than one node and you need all values under all nodes
- `key('tablename','lookup',$seq)` is invalid
- `$seq/key('tablename','lookup',.)` is acceptable
  - there is only one node being passed to the `key()` function at a time
  - returns all lookup values under all nodes in `$seq`
- similarly, if `$seq` was constrained to be optional, the same technique would prevent an error when the value was absent

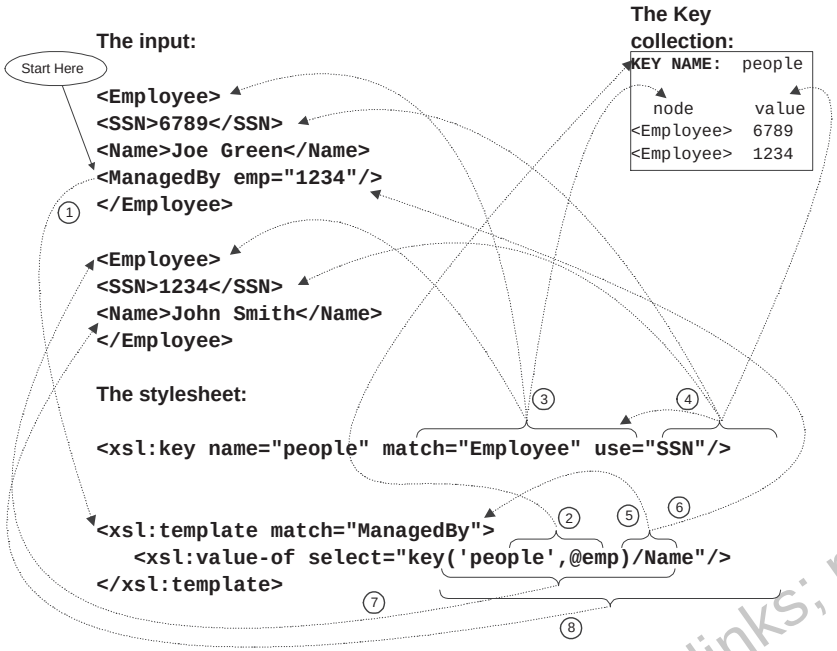
A separate set of key tables is maintained for each source document

- the table that is accessed is that for the document in which the current node is found

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT key node referencing (cont.)

The following illustrates how keys can be used to obtain the value of an employee's name indirectly from the value of an attribute (though another model might have the value in a child element's content) used to reference the employee information of a manager:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

XSLT key node referencing (cont.)

Chapter 7 - Data type expressions and functions
Section 4 - Node-set expressions

The following steps correspond to the diagram indicating how the value "John Smith" is obtained as the name of the manager of "Joe Green" indirectly through the `emp=` attribute of the `<ManagedBy>` element:

1. the template rule for `<ManagedBy>` needs to calculate a value from another node based on an attribute value in `emp=` (in another model it could just as well be based on an element node's value rather than an attribute node's value)
2. the target value calculation begins by first referencing the key named "`people`", using the first argument to the `key()` function, specifying which of the lookup key tables that were created by the XSLT processor based on values found in the source tree
3. the key named "`people`" is declared in a top-level `<xsl:key>` element building a table of nodes matching the given node-set expression indicated with `match=` (all element nodes named "`Employee`")
4. the `use=` attribute specifies the expression whose evaluated value, relative to each `match=` node, that is used for the lookup comparison value in the key table (the value of the first child element node named "`SSN`" relative to the element nodes named "`Employee`")
5. the second argument to `key()` function indicates the node (`emp=` attribute) relative to the current node (`ManagedBy` element) whose value is used for the search comparison with the key values (the attribute node for `emp=`)
6. the example indicates the string value "`1234`" is what is searched for among the key lookup values
7. `key()` returns all key nodes (`Employee` elements) whose lookup value is the string equivalent of the expression value for what is being searched for (only one node in this example)
8. the `<xsl:value-of>` is calculated by then evaluating the complete expression by obtaining the XPath expression (the child element node named "`Name`") relative to the returned set of key nodes (`Employee` elements), thus returning the resulting value (the string "John Smith")

Note that the evaluation of `key('people','1234')` would have produced the same result since the lookup value is the supplied string instead of an evaluated expression.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Current node referencing in XSLT

Referencing the current node from the start of an expression:

Remembering how the current node changes during expression evaluation, it is sometimes important in the middle of an expression to reference that node that was the current node before evaluation began.

`current()`

- re-orients within an XPath expression to the current node in the context in use at the start of the expression

Consider an example of comparing a remote node's attribute value to the attribute value of the current node. This can be accomplished using a variable to store the value of an attribute relative to the current node at the start of the expression evaluation for the `<xsl:for-each>` below:

- recall that after the `frame` node test, the context is changed to each `frame` element for the evaluation of the predicate

```
01 <xsl:template match="module">
02   <xsl:variable name="mod-label" select="@mod-label"/>
03   <xsl:for-each select="id(@other-module)//
04                         frame[@frame-label=$mod-label]">
05     <!--template-->
06   </xsl:for-each>
07 </xsl:template>
```

During the evaluation of the predicate, the expression evaluation can revert to the current node of the start of expression evaluation (the `module` element) as follows, thus producing the equivalent expression without the use of a variable declaration and assignment:

```
01 <xsl:template match="module">
02   <xsl:for-each select="id(@other-module)//
03                         frame[@frame-label=current()/@mod-label]">
04     <!--template-->
05   </xsl:for-each>
06 </xsl:template>
```

`current()` is different than the "`.`" abbreviation

- "`.`" represents the current node at the point of evaluation in the expression
- `current()` returns the current node at the start of evaluation of the expression

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sequence operator and functions

P2 operator `,`

- concatenate two sequences into a new sequence
- sequences are not nested
- e.g. "`(1,2,3),(4,5),(1,3)`" is "`(1,2,3,4,5,1,3)`"
- e.g. "`(1,2,3),(),(1,3)`" is "`(1,2,3,1,3)`"
- e.g. "(*`expr1`*,*`expr2`*)`[1]`" is "`if (exists(`*`expr1`*`)) then `*`expr1`* `else `*`expr2`*"
  - the abbreviated form calculates *`expr1`* only once
  - though it is up to the processor to decide if `expr2` is or is not evaluated in both cases
  - an optimizing processor may rewrite the expanded form as the abbreviated form
- the effective Boolean value of the sequence is conditional on its content
  - an empty sequence returns false
  - a singleton sequence returns the effective Boolean value of the singleton
  - a sequence beginning with a non-empty node set returns true
  - any other sequence of more than one item triggers an error

P2 `empty(`*`required-sequence`*`)`

- returns `true` if the sequence is empty and `false` if not
- one cannot use `boolean()` for this because of a singleton sequence evaluating to the Boolean of the singleton value

P2 `exists(`*`required-sequence`*`)`

- returns `false` if the sequence is empty and `true` if not
- one cannot use `not(boolean())` for this because of a singleton sequence evaluating to the Boolean of the singleton value

`count(`*`required-sequence`*`)`

- returns the number of items in the supplied sequence
- all nodes of any node set in the sequence are included in the count

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sequence operator and functions (cont.)

Cardinality assertions for static analysis:

- `zero-or-one(`*`required-sequence`*`)`
  - returns the supplied argument if the cardinality of the supplied argument satisfies the "optional" cardinality
  - useful for static checking before execution or runtime checking during execution
- `one-or-more(`*`required-sequence`*`)`
  - returns the supplied argument if the cardinality of the supplied argument satisfies the "mandatory and repeatable" cardinality
  - useful for static checking before execution or runtime checking during execution
- `exactly-one(`*`required-sequence`*`)`
  - returns the supplied argument if the cardinality of the supplied argument satisfies the "mandatory" cardinality
  - useful for static checking before execution or runtime checking during execution

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Sequence operator and functions (cont.)

Chapter 7 - Data type expressions and functions
Section 5 - Sequence expressions

- Sequence type operators (recall slide 74)
  - an expression can check the type of a sequence
    - `5 instance of xs:integer`
      - returns `true` because 5 is an `xs:integer` value
    - `5. instance of xs:integer`
      - returns `false` because the value isn't an integer
    - `(5, 6) instance of xs:integer+`
      - returns `true` because the sequence is one or more `xs:integer` values
    - `. instance of element()`
      - returns `true` if the current node is an element node
    - `. instance of usa:zipcode`
      - returns `true` if the current node is an instance of the user-defined type "`usa:zipcode`"
  - an expression can check the ability to convert the type of a singleton item
    - `5. castable as xs:integer`
      - returns `true` because the value could be cast as an integer
    - `'2007-08-32' castable as xs:date`
      - returns `false`
    - `'2007-08-31' castable as xs:date`
      - returns `true`
    - `$var castable as xs:integer`
      - returns true if the variable `$var` can be converted to an integer
    - `$addr castable as USAddress`
      - returns true if the variable `$addr` can be converted to "`USAddress`"
  - an expression can convert the type of a singleton item
    - `$var cast as xs:integer`
      - returns the `$var` variable as an integer value
    - `$addr cast as USAddress`
      - returns the `$addr` variable as having the type "`USAddress`"
    - `zip cast as usa:zipcode` or `usa:zipcode( zip )`
      - returns the `zip` child of the current node as having the type "`usa:zipcode`"
  - an expression can assert the type of a singleton item
    - `$var treat as xs:integer`
      - only if the variable `$var` is an instance of the given type, say "`xs:integer`", then the variable's value is returned, otherwise a run-time error is generated
      - the value is *not* cast in order to be returned without error
  - a nuance when dealing with qualified names
    - `'abc:def' cast as xs:QName`
      - returns true if the given string has a comparison value in the transform context
      - returns false for all variables and other strings

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Sequence operator and functions (cont.)

- `distinct-values(`*`required-sequence`*`)`
- `distinct-values(`*`required-sequence`*`,`*`collation`*`)`
  - returns the members of the sequence with duplicates removed based on the "eq" test
  - supplying a collation overrides the default collation for evaluated string values
  - all occurrences of `NaN` are reduced to a single `NaN`
    - even though `NaN != NaN`
- `deep-equal(`*`sequence`*`,`*`sequence`*`)`
- `deep-equal(`*`sequence`*`,`*`sequence`*`,`*`collation`*`)`
  - returns `true` if the two arguments are identical sequences accounting for the member types and, if nodes, the complete member element structures from each apex to the leaves of the tree
  - supplying a collation overrides the default collation for string values (but not node names)
- `index-of(`*`search-sequence`*`,`*`lookup-singleton`*`)`
- `index-of(`*`search-sequence`*`,`*`lookup-singleton`*`,`*`collation`*`)`
  - returns a sequence of integers representing the indexes within the search sequence where the lookup singleton is found
  - e.g. `index-of((10, 20, 30, 30, 20, 10), 20)` returns `(2, 5)`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sequence operator and functions (cont.)

- `insert-before(`*`first-sequence`*`,`*`integer`*`,`*`second-sequence`*`)`
  - inserts the second sequence into the first sequence before the member of the first sequence indicated by the integer
  - counting starts at one and numbers below one and above the count of members will insert the second sequence respectively at the beginning and end of the first sequence
- `remove(`*`sequence`*`,`*`integer`*`)`
  - removes the member of the sequence indicated by the integer
- `reverse(`*`sequence`*`)`
  - returns the members of the input sequence in a sequence in reverse order as input
  - e.g. `reverse( 5 to 7 )` returns `( 7, 6, 5 )`
- `subsequence(`*`sequence`*`,`*`start-integer`*`)`
- `subsequence(`*`sequence`*`,`*`start-integer`*`,`*`count-integer`*`)`
  - returns the members of the input sequence starting at the second argument for the count of members indicated in the third argument or to the end of the sequence if the third argument is absent
- `unordered(`*`items`*`)`
  - returns the members of the input items in an arbitrary order
  - if the items are selected by a node set address this might provide an optimization by not having to order the members addressed in document order

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Sequence operator and functions (cont.)

P2 `max(`*`required-sequence`*`)` and `max(`*`required-sequence`*`,`*`collation`*`)`
- returns the maximum value found in the sequence
- optionally override the default collation when dealing with strings

P2 `min(`*`required-sequence`*`)` and `min(`*`required-sequence`*`,`*`collation`*`)`
- returns the minimum value found in the sequence
- optionally override the default collation when dealing with strings

P2 `avg(`*`required-sequence`*`)`
- returns the average of values (sum divided by count)

`sum(`*`required-sequence`*`)`
- converts each node to a string before converting to a number
- calculates and returns the sum of each of the number values
- returns an integer zero for the empty sequence of values

For arithmetic on sets use a predicate to ensure all of the values addressed are suitable:
- need a predicate that is true for numbers but false for `NaN`
- P1 e.g. `sum(`*`path`*`[number()=number()])`
  - this relies on a comparison operator returning false when `NaN` is used
- P2 e.g. `sum(`*`path`*`[. castable as xs:double])`
  - this relies on string values being implicitly cast to `xs:double`
- P2 e.g. `sum(`*`path`*`[. castable as xs:dayTimeDuration]/ xs:dayTimeDuration(.))`
  - this avoids the implicit cast to `xs:double` by explicitly casting to `xs:dayTimeDuration`
  - note that when the node set summed is empty, this still returns the `xs:double` value of 0, not an `xs:dayTimeDuration` value

P2 `sum(`*`required-sequence`*`,`*`empty-sequence-zero-value`*`)`
- same as `sum()` but when the required sequence in the first argument has no items or nodes then the second argument is returned as the zero value
- important when the required return value type cannot be inferred from an integer zero, e.g. in the following the variable's type constraint requires `sum()` to return a value of type `xs:daytimeDuration` when there are no nodes being summed:
- ```
  01 <xsl:variable name="waitTime" as="xs:dayTimeDuration"
  02  select="sum($nodes[. castable as xs:dayTimeDuration]/
  03              xs:dayTimeDuration(.), xs:dayTimeDuration('PT0S'))"/>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using Boolean functions

Boolean operators and functions:
- the comparison space is only `true` and `false`
- the lexical space is only "`1`" and "`true`" for true and "`0`" and "`false`" for false

`boolean(`*`required-expression`*`)`
- converts the essence of the argument to a Boolean value
- converts an empty string to `false` and all other strings to `true`
- converts NaN and positive and negative 0 to `false` and all other values to `true`
  - `NaN` is not equal to anything (including itself), so to test that a value is `NaN`:
    - `test="not( number($num) = number($num) )"`
- P1 converts an empty node list to `false` and all other node lists to `true`
- P2 converts an empty sequence to `false`, a singleton sequence according to the singleton's data type, and longer sequences trigger a runtime error
- converts any result-tree fragment or temporary tree to `true` because of the document node
- different than the casting to the data type using `xs:boolean(`*`expression`*`)`
  - the expression for casting must be in the lexical space

`true()` and `false()`
- return the values `true` and `false`

`not(`*`expression`*`)`
- returns the opposite after converting the expression using `boolean()`

`lang(`*`expression`*`)`

P2 `lang(`*`expression`*`, `*`node`*`)`
- supports XML 1.0 Section 2.12 Language Identification
  - `xml:lang="`*`primary`*`"` or `xml:lang="`*`primary`*`-`*`sub`*`"`
    - e.g. `lang="en"`, `lang="en-US"`, `lang="en-cockney"`
  - governed by IETF Request For Comment 3066 (making RFC 1766 obsolete)
  - one primary component and any number (including zero) subcomponents
    - subcomponents are only administrative
- returns `true` if an ancestral element to the supplied node (or context node if a second argument is not supplied) has an attribute named "`xml:lang`" whose value, or whose initial portion up to the first embedded "-", is equal to the case-insensitive expression
- returns `false` if no such ancestral attribute is found

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.
  ```

Calculating values using Boolean functions (cont.)

operators `or` and `and`

- returns traditional results of Boolean logic
- when evaluating the `or` operator, evaluation terminates at the first operand evaluating to `true`
  - note that the "`|`" operator is the union operator and not the logical `or` as is popular in many programming languages
- when evaluating the `and` operator, evaluation terminates at the first operand evaluating to `false`

grouping `(` and `)`

- performs the traditional grouping to override precedence

P2 operators "`<<`", "`is`" and "`>>`" (not including the quotes):

- referred to as "node comparisons"
- tests on singleton node operands being in document order
- returns respectively whether the left-hand operand is before the right-hand operand in document order, is the same node as the right-hand operator, or is after the right-hand operator in document order
- the "<" must always be escaped (e.g. using "`<`")

P2 operators "`eq`", "`ne`", "`gt`", "`lt`", "`le`" and "`ge`" (not including the quotes):

- referred to as "value comparisons"
- returns `true` or `false` on the comparison of singleton operand values

operators "<", ">", "<=", ">=", "=" and "`!=`" (not including the quotes):

- the "<" must always be escaped (e.g. using "`<`")
- referred to as "general comparisons"
- P1 if "<" or ">" used in the operator, both operands converted using `number()`
- returns `true` or `false` on the comparison of arbitrary operand values
  - each operand may be a singleton or a sequence or a node set
- two Boolean operands
  - considered equal if their respective values are equal
- two number operands
  - considered equal if their respective values are equal
- two string operands
  - P1 considered equal if their Unicode characters are identical
    - the full form and composed form of a given character are not identical
  - P2 considered equal by the default collation determination in play
- two singleton operands
  - P2 convert unlike operands to like objects before performing the comparison
    - P1 uses `boolean()` if either is Boolean, otherwise uses `number()`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Calculating values using Boolean functions (cont.)

Chapter 7 - Data type expressions and functions
Section 6 - Boolean expressions

operators "<", ">", "<=", ">=", "=" and "`!=`" (not including the quotes) (cont.):

- one or two node-set or sequence operands
  - the result of the comparison is initialized to `false` before beginning the comparison
  - `false` is returned when either operand is empty
    - "`a != b`" and "`not( a = b )`" and "`a ne b`" are all different
    - "`a != b`" will return false if either operand does not exist
    - "`not( a = b )`" will return true if either operand does not exist
    - "`a ne b`" will trigger an error if either operand does not exist
    - all three expressions return the same when both operands exist as singletons
  - deal with values of each individual member of the node set until `true`
  - the operators performed on two node sets will consider the number values (if greater-than or less-than is used in the operator) or the string values of the members of the node sets and consider the operation `true` if the value of any member of one node set compares as desired to the value of any member of the other node set and `false` if no member of one set compares as desired to all values of the other set
  - the operators performed on a node set and a singleton argument convert the node set member values to like objects (the precedence being number if greater-than or less-than is used in the operator, then Boolean if the singleton argument is a Boolean, then number if it is a number, otherwise string) before performing the comparisons and consider the operation `true` if the value of any member of the node set compares as desired to the value of the other argument
  - note that this means that the Boolean `not()` function applied to a node set comparison will return `true` if all members of the set fail the comparison
  - does *not* test for two nodes as being the same node
    - because comparisons are based on the value of the node not the identity of the node

- `test='switch/@state="on"'`
  - `true` if *any* of the children "`switch`" have the attribute equal to "`on`"
  - the test checks each of the nodes until the first `true` comparison and only returns `false` if all nodes test `false`
  - P2 `test="some $s in switch/@state satisfies $s eq 'on'"`
- `test='not(switch/@state!="on")'`
  - `true` if *all* of the children "`switch`" have the attribute equal to "`on`"
  - the inner test checks each of the nodes and only returns `false` if all nodes test `false` (that is that they all have the value "`on`")
  - the outer evaluation changes the inner evaluation from `false` to `true`
  - P2 `test="every $s in switch/@state satisfies $s eq 'on'"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Calculating values using Boolean functions (cont.)

P2 Convenience in XPath 2 for multiple "`or`" comparisons:

- creating a sequence of comparison values and using the symbolic comparison operators will require the processor to walk all of the members of the sequence until one of them compares as "`true`"
- when all of the members are compared without a "`true`" response then the expression returns "`false`"

```
01 question[@response=( 'y', 'Y', 'n', 'N' )]
```

- the above expression in XPath 1.0 would require a four-operand "`or`" expression

```
01 ulink[lower-case(substring-before(normalize-space(@url),':'))=
02               ( 'ftp', 'http', 'https', 'mailto' )]
```

- the above template match for a DocBook construct in XPath 1.0 would also require a four-operand "`or`" expression with multiple invocations of the function calls

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Calculating values using Boolean functions (cont.)

P2 XPath 2.0 standalone quantified expressions

- an expression returning a Boolean `true` or `false` based on the presence of one `true` value or all `true` values in a set of comparisons
- the tuple can have as many members as is needed to write the comparison
- the following returns `true`

```
- some $x in (1, 2, 3), $y in (4, 5) satisfies $x + $y eq 6
```

- the following returns `false`

```
- every $x in (1, 2, 3), $y in (4, 5) satisfies $x + $y eq 6
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Qualified-name functions

Chapter 7 - Data type expressions and functions
Section 7 - Miscellaneous expressions

Recall various functions that return node name information as strings:

- see page 308 for string-based name functions
- recall page 30 with terminology

QName values have different lexical and comparison spaces

- the lexical value is either a prefixed name or an un-prefixed qualified name used for reference and for exposition
- the comparison value is the expanded name
- the namespace name for un-prefixed attributes is the empty string
- the namespace name for un-prefixed elements is the default namespace URI

P2 `QName(`*`string-uri`*`,`*`string-qname`*`)`

- returns the qualified name value using the second argument with the URI of the supplied prefix (or default namespace if not present) being the first argument

Dealing with both the namespace name (URI) and local name as a unit value

- P2 e.g. `count(ancestor::*[node-name(.)=QName('urn:X-p','a')])`
- count the ancestor elements named "a" in the "`urn:X-p`" namespace
- no heed of or hindrance from choices of prefixed or non-prefixed names

P2 `prefix-from-QName(`*`qname`*`)`

- returns the non-colon prefix string of the interpretation of the supplied qualified name value

P2 `local-name-from-QName(`*`qname`*`)`

- returns the non-colon local name string of the interpretation of the supplied qualified name value

P2 `namespace-uri-from-QName(`*`qname`*`)`

- returns the URI value of the interpretation of the supplied qualified name value

P2 `resolve-QName(`*`string-qname`*`,`*`element-node`*`)`

- returns the qualified name value of the interpretation of the supplied string as a qualified name in the context of the supplied element node

Tip:

- sometimes it is necessary to construct a qualified name ignoring the default namespace
  - cannot just use `resolve-QName()` because it respects the default namespace
- `if (contains($qname, ":"))`
  `then resolve-QName($qname, $element)`
  `else QName("", $qname)`
  - when there is a prefix present, determine the name in the context of an element
  - otherwise, ignore the default namespace and construct a name using no namespace

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Qualified-name functions (cont.)

Chapter 7 - Data type expressions and functions
Section 7 - Miscellaneous expressions

P2 `in-scope-prefixes(`*`element-node`*`)`

- returns a sequence of strings reflecting the prefixes of all of the in-scope namespaces for the given element

P2 `namespace-uri-for-prefix(`*`prefix-string`*`,`*`element-node`*`)`

- returns the URI (not a string) of the interpretation of the supplied qualified name (not a string)

Tip:

- when the namespace axis is unavailable, these functions can be used
  - e.g. `name(namespace::*[.=$nsuri])`
  - replaced with:
  - `in-scope-prefixes($node)`
    `[namespace-uri-for-prefix(.,$node)=$nsuri]`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

URI functions

P2 `resolve-uri(`*`abs-or-relative-new-uri`*`,`*`abs-or-relative-base-uri`*`)`

- returns the URI of the first argument string interpreted as an absolute or relative URI against the second string interpreted as the base URI (absolute or relative)
- note the return is a URI and not a string
- if both URI strings are relative, then the return URI is relative
- e.g. `resolve-uri('abc.xml','def/ghi/jkl.xml')`
  - returns the relative URI "`def/ghi/abc.xml`"
- e.g. `resolve-uri('abc.xml',base-uri(.))`
  - returns the absolute URI of "`abc.xml`" using in the same directory as that for the fragment from which the current source tree node was obtained

P2 `encode-for-uri(`*`string`*`)`

- returns the argument string encoded according to the rules of RFC 3986
  - `http://www.ietf.org/rfc/rfc3986.txt`
  - characters other than A-Z, 0-9, "-", "_", "." and "~" are encoded using UTF-8 and then escaped with "%"
- e.g. `encode-for-uri("http://www.example.com/~bébé")`
  returns "`http%3A%2F%2Fwww.example.com%2F~b%C3%A9b%C3%A9`"
- the entire string is processed, so it only practical to work on only portions of an address
- e.g. `concat("http://www.example.com/", encode-for-uri("50% off"))`
  returns "`http://www.example.com/50%25%20off`"

P2 `iri-to-uri(`*`string`*`)`

- returns the argument string encoded according to the rules of RFC 3987
  - `http://www.ietf.org/rfc/rfc3987.txt`
  - converts non-URI characters to escaped UTF-8 sequences
- e.g. `iri-to-uri("http://www.example.com/50% off - bébé")`
  returns "`http://www.example.com/50%%20off%20-%20b%C3%A9b%C3%A9`"

P2 `escape-html-uri(`*`string`*`)`

- returns the argument string escaped for all non-ASCII characters
  - converts to escaped UTF-8 sequences
- e.g. `escape-html-uri("http://www.example.com/50% off - bébé")`
  returns "`http://www.example.com/50% off - b%C3%A9b%C3%A9`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Date and time functions and operators

P2 Comparison values for different date and time types defined in the data model

- `http://www.w3.org/TR/xpath-datamodel/#dates-and-times`
- data types defined in detail in the W3C Schema specification
  - `http://www.w3.org/TR/xmlschema-2/#isoformats`
- XPath 2.0 introduces two subtypes of duration not included in W3C Schema
  - see Data types (page 73)

Date and time lexical values

- "`-`*`yyyy`*`-`*`mm`*`-`*`dd`*`T`*`hh`*`:`*`mm`*`:`*`ss`*" (BCE)
- "*`yyyy`*`-`*`mm`*`-`*`dd`*`T`*`hh`*`:`*`mm`*`:`*`ss`*" (CE)
- "*`yyyy`*`-`*`mm`*`-`*`dd`*`T`*`hh`*`:`*`mm`*`:`*`ss.ssss±hh:mm`*"
- "*`yyyy`*`-`*`mm`*`-`*`dd`*`T`*`hh`*`:`*`mm`*`:`*`ss.ssss`*`Z`" (UTC)
- either the date or the time or both can be specified
  - '`T`' is only used when both are specified
- no limit to the fractional seconds
- '`Z`' must be presented in a value with a time zone difference from UTC
- example values:
  - "`17:23`" represents 5:23pm in the implicit time zone
  - "`17:23-04:00`" represents 5:23pm in Eastern Daylight Savings (EDT) time
  - "`2007-08-17-04:00`" represents August 17, 2007 EDT
  - "`2001-01-01T00:00:00+00:00`" represents midnight at the start of the millennium

Cast strings to date/time using the primitive types:

- `xs:dateTime('2007-08-17T21:43:00')`
- `xs:date('2007-08-17')`
- `xs:time('21:43:00')`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Date and time functions and operators (cont.)

Duration lexical values

- "P*n*Y*n*M*n*DT*n*H*n*M*n*S"
- "P*n*Y*n*M*n*DT*n*H*n*M*n.nnn*S"
- the "P" is required (historically referred to as "period")
- any component may be absent provided not all components are absent
  - a component's letter must be accompanied by its number value
- the "T" is only used when there are both date and time components
- *n* may be the value zero
- XPath 2.0 `xs:yearMonthDuration`
  - "P*n*Y*n*M"
  - stored month value is limited to between 0 and 11
  - specified values for casting can be any number
- XPath 2.0 `xs:dayTimeDuration`
  - "P*n*DT*n*H*n*M*n.nnn*S"
  - stored hour, minute and second values are limited not to be larger than the next unit of measure
  - specified values for casting can be any number
- example values:
  - "P0M" represents zero months (canonical zero `xs:yearMonthDuration`)
    - if all of the components are zero
  - "PT0S" represents zero seconds (canonical zero `xs:dayTimeDuration`)
    - if all of the components are zero
  - "PT47H" represents 47 hours
  - "P1DT1M" represents one day and 1 minute

Cast strings to date/time using the type name as a function or using the operator:

- `xs:duration('P1Y3D')`
- `xs:yearMonthDuration('P1Y2M')`
- `'P3DT10H20S' cast as xs:dayTimeDuration`

Arithmetic only allowed for derived XPath durations, not XSD durations

- `xs:duration` is defined by XSD
- `xs:yearMonthDuration` and `xs:dayTimeDuration` are derived in XPath from `xs:duration` in XSD

operator `-` on dates and times returns derived durations

operators `-` and `+` on dates and times with derived durations returns dates and times

operators `*` and `div` on derived durations with scalars returns derived durations

operator `div` on two derived durations returns a scalar

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Date and time functions and operators (cont.)

`current-dateTime()`

- returns the `dateTime` expression of the current date and time
- for the duration of executing the transform this returns a stable single value

`current-date()`

- returns the `date` expression of the current date
- for the duration of executing the transform this returns a stable single value

`current-time()`

- returns the `time` expression of the current time
- for the duration of executing the transform this returns a stable single value

`implicit-timezone()`

- returns the `dayTimeDuration` expression of the implicit time zone
- e.g. Eastern Daylight Savings Time returns "`-PT4H`"

`dateTime(`*`date?`*`,`*`time?`*`)`

- returns the `dateTime` expression of the date and time
- either one may be the empty sequence, in which case the return is the empty sequence

`adjust-dateTime-to-timezone(`*`dateTime`*`,`*`optional-tz-dayTimeDuration`*`)`

- omitting the time zone value returns the `dateTime` in the implicit time zone
- an empty sequence time zone value returns the `dateTime` without a time zone
- a time zone value returns the `dateTime` in the time zone

`adjust-date-to-timezone(`*`date`*`,`*`optional-tz-dayTimeDuration`*`)`

- assumes the time is `00:00:00` for calculations
- omitting the time zone value returns the `date` in the implicit time zone
- an empty sequence time zone value returns the `date` without a time zone
- a time zone value returns the `date` in the time zone

`adjust-time-to-timezone(`*`time`*`,`*`optional-tz-dayTimeDuration`*`)`

- omitting the time zone value returns the `time` in the implicit time zone
- an empty sequence time zone value returns the `time` without a time zone
- a time zone value returns the `time` in the time zone

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Date and time functions and operators (cont.)

- `year-from-dateTime(`*`dateTime`*`)`
  - returns the integer year from the date and time
- `month-from-dateTime(`*`dateTime`*`)`
  - returns the integer month from the date and time
- `day-from-dateTime(`*`dateTime`*`)`
  - returns the integer day from the date and time
- `hours-from-dateTime(`*`dateTime`*`)`
  - returns the integer hours from the date and time
- `minutes-from-dateTime(`*`dateTime`*`)`
  - returns the integer minutes from the date and time
- `seconds-from-dateTime(`*`dateTime`*`)`
  - returns the integer seconds from the date and time
- `timezone-from-dateTime(`*`dateTime`*`)`
  - returns the duration of the time zone from the date and time

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Date and time functions and operators (cont.)

- `year-from-date(`*`date`*`)`
  - returns the integer year from the date
- `month-from-date(`*`date`*`)`
  - returns the integer month from the date
- `day-from-date(`*`date`*`)`
  - returns the integer day from the date
- `timezone-from-date(`*`date`*`)`
  - returns the duration of the time zone from the date
- `hours-from-time(`*`time`*`)`
  - returns the integer hours from the time
- `minutes-from-time(`*`time`*`)`
  - returns the integer minutes from the time
- `seconds-from-time(`*`time`*`)`
  - returns the integer seconds from the time
- `timezone-from-time(`*`time`*`)`
  - returns the duration of the time zone date and time

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Date and time functions and operators (cont.)

- `years-from-duration(`*`duration`*`)`
  - returns the integer year from the duration
- `months-from-duration(`*`duration`*`)`
  - returns the integer month from the duration
- `days-from-duration(`*`duration`*`)`
  - returns the integer day from the duration
- `hours-from-duration(`*`duration`*`)`
  - returns the integer hours from the duration
- `minutes-from-duration(`*`duration`*`)`
  - returns the integer minutes from the duration
- `seconds-from-duration(`*`duration`*`)`
  - returns the integer seconds from the duration

Important nuances regarding durations

- extracting a component extracts only the component
  - e.g. "`seconds-from-duration(xs:duration('PT2H3S'))`" returns `3`
- arithmetic only allowed on duration subtypes
  - e.g. "`xs:dayTimeDuration('PT2H3S') div xs:dayTimeDuration('PT1S')`" returns `7203`
  - e.g. "`xs:duration('PT2H3S') div xs:duration('PT1S')`" returns an error
    - see page 73 for subtypes

Obtaining only the fractional part of seconds

- recall that the `mod` operator gives the remainder after whole division
- `seconds-from-time(current-time()) mod 1`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Formatting date and time strings

- `format-dateTime(`*`dateTime`*`, `*`pattern`*`)`
- `format-dateTime(`*`dateTime`*`, `*`pattern`*`, `*`language`*`, `*`calendar`*`, `*`country`*`)`
- `format-date(`*`date`*`, `*`pattern`*`)`
- `format-date(`*`date`*`, `*`pattern`*`, `*`language`*`, `*`calendar`*`, `*`country`*`)`
- `format-time(`*`time`*`, `*`pattern`*`)`
- `format-time(`*`time`*`, `*`pattern`*`, `*`language`*`, `*`calendar`*`, `*`country`*`)`
  - returns the string representation of the given time, formatted according to the given picture string
  - e.g. `format-date($d,"[Y0001]-[M01]-[D01]")` returns `2007-08-26`
  - e.g. `format-date($d,"[Y]-[M]-[D]")` returns `2007-8-26`
  - e.g. `format-date($d, "[D01] [MN,*-3] [Y0001]", "en", (), ())` returns `26 AUG 2007`
  - e.g. returning `2007-08-26 14:05z`

```
format-dateTime(
  adjust-dateTime-to-timezone(current-dateTime(),
                              xsd:dayTimeDuration('PT0H')),
  '[Y0001]-[M01]-[D01] [H01]:[m01]z')
```

    - the current time is converted to Coordinated Universal Time/Temps Universel Coordonné (UTC) before formatting
      - time zone "00:00"
      - also called "Zulu time"
  - the `language`, `calendar` and `country` arguments available are implementation-defined
    - defaults are also implementation-defined
    - impacts on words used for names (e.g. months, days of the week)
    - impacts on the ordinals (e.g. `20th`)
    - impacts on the convention for expressing hours
    - impacts on the first day of the week and first day of the year

Full specification of features in the XSLT 2.0 recommendation:

- `http://www.w3.org/TR/xslt20/#format-date`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Formatting date and time strings (cont.)

Quick reference to pattern strings:

`Y`	year (absolute value)
`M`	month in year
`D`	day in month
`d`	day in year
`F`	day of week
`W`	week in year
`w`	week in month
`H`	hour in day (24 hours)
`h`	hour in half-day (12 hours)
`P`	am/pm marker
`m`	minute in hour
`s`	second in minute
`f`	fractional seconds
`Z`	time zone as a time offset from UTC, or alphabetic (e.g. "`EST`")
`z`	time zone as a time offset using GMT (e.g. "`GMT+1`")
`C`	calendar: the name or abbreviation of a calendar name
`E`	era: the name of a baseline for the numbering of years
`token`	modifier: see Formatting numbers as a sequence of characters (page 399)
`n`	modifier: name in lower case
`N`	modifier: name in upper case
`Nn`	modifier: name in mixed case
`t`	modifier: traditional (vs. alphabetic)
`o`	modifier: ordinal
`,`	modifier: length "`,`*`min-width-or-*`*`-`*`max-width`*""

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Inferring structure when there is none

An XPath location step addresses all nodes along an axis before being filtered by a predicate.

Consider the need to infer nested structure from a flat structure when formatting lists and list items from the instance `bullet.xml`:

```
<?xml version="1.0"?>
<doc>
<listhead>The first list of information:</listhead>
<bullet>first member of first list</bullet>
<bullet>second member of first list</bullet>
<bullet>third member of first list</bullet>
<listhead>The second list of information:</listhead>
<bullet>first member of second list</bullet>
<bullet>second member of second list</bullet>
</doc>
```

The desired output is:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<p>The first list of information:</p>
<ul>
<li>first member of first list</li>
<li>second member of first list</li>
<li>third member of first list</li>
</ul>

<p>The second list of information:</p>
<ul>
<li>first member of second list</li>
<li>second member of second list</li>
</ul>

</html>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Inferring structure when there is none (cont.)

It is necessary to deal with sibling elements in order to place adjacent siblings into the inferred structure, but XPath expressions do not distinguish adjacent siblings from siblings that are separated from each other.

Note the differences between the following expressions available with XPath:

- `following-sibling::bullet`
  - all following sibling element nodes named "`bullet`"
- `following-sibling::*[self::bullet]`
  - all following sibling element nodes named "`bullet`"
- `following-sibling::bullet[1]`
  - the first following sibling element nodes named "`bullet`", not considering that there may or may not be any intervening sibling elements of other types
- `following-sibling::*[1]`
  - the immediately following sibling element node, regardless that node's name
- `following-sibling::*[1][self::bullet]`
  - the immediately following sibling element node *only* if it is named "`bullet`", otherwise the empty node set is returned

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Inferring structure when there is none (cont.)

The following `bullet.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--bullet.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output indent="yes"/>

<xsl:template match="doc">
  <html><xsl:apply-templates/></html>
</xsl:template>

<xsl:template match="bullet"> <!--handle every bullet in turn-->
  <xsl:choose><!--don't always act on *every* bullet there is-->
        <!--when there is an immediate previous sibling of the
            same name, then assume this node has already been
            addressed by the first of the contiguous siblings-->
    <xsl:when test="preceding-sibling::*[1][self::bullet]"/>
    <xsl:otherwise><!--at the first of a group of siblings-->
      <ul><!--"walk" along all sibling bullets for list items-->
        <xsl:apply-templates select="." mode="sibling-bullets"/>
      </ul>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

        <!--place each of a group of adjacent siblings-->
<xsl:template match="bullet" mode="sibling-bullets">
  <li><xsl:apply-templates/></li> <!--put out this one-->
                                  <!--go to next one-->
  <xsl:apply-templates mode="sibling-bullets"
            select="following-sibling::*[1][self::bullet]"/>
</xsl:template>

<xsl:template match="listhead">   <!--other stuff-->
  <p><xsl:apply-templates/></p>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Templates as pseudo-subroutines

Chapter 7 - Data type expressions and functions
Section 9 - Traversing the source tree

Some algorithms required for transformation are not met by simple pattern matching and node set selection when:

- sibling elements are indistinguishable to pattern matching
- ancestral elements are indistinguishable to pattern matching
- it is sometimes necessary to "walk" along the source node tree to find or process information

By passing control to another template rule, opportunities exist to:

- add to the result tree before passing control again
- add to the result tree after control is passed back
- pass values to bind to parameterized variables

Consider the (contrived) need to process the following `invert.xml` instance:

```
<?xml version="1.0"?>
<test val="a">
  <test val="b">
     <test val="c">
        <test val="d">
          <test>
            <greeting>Greeting in the middle of the file.</greeting>
          </test>
        </test>
     </test>
  </test>
</test>
```

Note how the innermost `test` element has no attribute, has no `test` element as a child, and is contained within elements that are nested back to the document element, each uniquely identified by an attribute value for reporting purposes (but not used in this contrived example for pattern matching).

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Templates as pseudo-subroutines (cont.)

Chapter 7 - Data type expressions and functions
Section 9 - Traversing the source tree

The following result echoes the instance with typical template rules and then inverts the instance inside-out, translating the innermost element, and then inverting the wrapping elements by walking the node tree towards the root (note the attributes are copied into both the mock start and end tags to illustrate where in the node tree processing is being executed):

```
<result>
Non-inverted Structure (translated):
{test val="a"}
{test val="b"}
{test val="c"}
{test val="d"}
{test}
<greeting>Greeting in the middle of the file.</greeting>
{/test}
{/test val="d"}
{/test val="c"}
{/test val="b"}
{/test val="a"}

Inverted Structure:
{/test}
{/test val="d"}
{/test val="c"}
{/test val="b"}
{/test val="a"}

{test val="a"}
{test val="b"}
{test val="c"}
{test val="d"}
{test}
</result>
```

Note how in the resulting inverted structure:

- the mock end tags precede the mock start tags
- the innermost `test` mock element (with attribute "a") is the original document element

This translation cannot be accomplished using only match patterns.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Templates as pseudo-subroutines (cont.)

The following stylesheet `invert-simple.xsl` walks the input structure:

```
<?xml version="1.0"?><!--invert-simple.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xA;">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output omit-xml-declaration="yes"/>

<xsl:template match="/">
 <result>
 <xsl:text>&nl;Non-inverted Structure (translated):</xsl:text>
 <xsl:apply-templates/>
 <xsl:text>&nl;Inverted Structure:&nl;</xsl:text>
           <!-- start from the test node that has no children-->
 <xsl:apply-templates mode="walk-up" select="//test[not(test)]"/>
 <xsl:text>&nl;</xsl:text>
 </result>
</xsl:template>

<xsl:template match="test">         <!--walking down the tree-->
  <xsl:text>&nl;{test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}</xsl:text>
  <xsl:apply-templates select="*"/>  <!--walk down to children-->
  <xsl:text>{/test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}&nl;</xsl:text>
</xsl:template>

<xsl:template match="greeting">         <!--preserve greeting-->
  <xsl:text>&nl;</xsl:text>
  <xsl:copy-of select="."/>
  <xsl:text>&nl;</xsl:text>
</xsl:template>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Templates as pseudo-subroutines (cont.)

```
<xsl:template mode="walk-up" match="*"><!--walking up the tree-->
                    <!--processing before the "subroutine call"-->
  <xsl:text>{/test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}&nl;</xsl:text>
  <xsl:if test="parent::*">
                       <!--the pseudo "subroutine call" itself-->
    <xsl:apply-templates mode="walk-up" select="parent::*"/>
  </xsl:if>
                     <!--processing after the "subroutine call"-->
  <xsl:text>&nl;{test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}</xsl:text>
</xsl:template>

<xsl:template name="showattrs">     <!--display the attributes-->
  <xsl:for-each select="@*">
    <xsl:text> </xsl:text>
    <xsl:value-of select="name(.)"/>="<xsl:value-of
                  select="."/>"</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Passing variables to pseudo-subroutines

The following result from the same data can be achieved by passing information when treating templates as pseudo-subroutines:

```
<result>
Non-inverted Structure (translated):
{test val="a"}
{test val="b"}
{test val="c"}
{test val="d"}
{test}<hello>Greeting in the middle of the file.</hello>{/test}
{/test val="d"}
{/test val="c"}
{/test val="b"}
{/test val="a"}

Inverted Structure:
{/test}
{/test val="d"}
{/test val="c"}
{/test val="b"}
{/test val="a"}
<hello>Greeting in the middle of the file.</hello>
<greeting>Greeting in the middle of the file.</greeting>
{test val="a"}
{test val="b"}
{test val="c"}
{test val="d"}
{test}
</result>
```

Information is passed from being generated in the innermost-nested source tree element, to be displayed in the outermost-nested source tree element that is the innermost result tree element. Both result tree fragments and node sets are shown to compare how their processing differs.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Passing variables to pseudo-subroutines (cont.)

The stylesheet `invert-parm.xsl` illustrates this technique of passing information:

```
<?xml version="1.0"?><!--invert-parm.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xA;">]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output omit-xml-declaration="yes"/>

<xsl:template match="/">
  <result>
  <xsl:text>&nl;Non-inverted Structure (translated):</xsl:text>
  <xsl:apply-templates/>
  <xsl:text>&nl;Inverted Structure:&nl;</xsl:text>
            <!-- start from the test node that has no children-->
  <xsl:apply-templates mode="walk-up" select="//test[not(test)]">
                            <!--illustrate difference between
                          result tree fragments and node sets-->
    <xsl:with-param name="middle-rtf">
      <xsl:apply-templates select="//test[not(test)]/*"/>
    </xsl:with-param>
    <xsl:with-param name="middle-node"
                    select="//test[not(test)]/*"/>
  </xsl:apply-templates>
  <xsl:text>&nl;</xsl:text>
  </result></xsl:template>

<xsl:template match="test">         <!--walking down the tree-->
  <xsl:text>&nl;{test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}</xsl:text>
  <xsl:apply-templates select="*"/>  <!--walk down to children-->
  <xsl:text>{/test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}&nl;</xsl:text></xsl:template>

<xsl:template match="greeting">        <!--transform greeting-->
  <hello><xsl:apply-templates/></hello></xsl:template>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Passing variables to pseudo-subroutines (cont.)

Chapter 7 - Data type expressions and functions
Section 9 - Traversing the source tree

(cont.)

```
<xsl:template mode="walk-up" match="*"><!--walking up the tree-->
  <xsl:param name="middle-rtf"/>       <!--"called" variables-->
  <xsl:param name="middle-node"/>
                  <!--processing before the "subroutine call"-->
  <xsl:text>{/test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}&nl;</xsl:text>
  <xsl:choose>
    <xsl:when test="parent::*">
                      <!--the pseudo "subroutine call" itself-->
     <xsl:apply-templates mode="walk-up" select="parent::*">
                                        <!--"calling" vars-->
       <xsl:with-param name="middle-rtf" select="$middle-rtf"/>
       <xsl:with-param name="middle-node" select="$middle-node"/>
     </xsl:apply-templates>
    </xsl:when>
    <xsl:otherwise>     <!--illustrate both types of variables-->
     <xsl:copy-of select="$middle-rtf"/>     <!--tree fragment-->
     <xsl:text>&nl;</xsl:text>
     <xsl:copy-of select="$middle-node"/>        <!--node set-->
    </xsl:otherwise>
  </xsl:choose>
                    <!--processing after the "subroutine call"-->
  <xsl:text>&nl;{test</xsl:text>
  <xsl:call-template name="showattrs"/>        <!--echo attrs-->
  <xsl:text>}</xsl:text>
</xsl:template>

<xsl:template name="showattrs">     <!--display the attributes-->
  <xsl:for-each select="@*">
    <xsl:text> </xsl:text>
    <xsl:value-of select="name(.)"/>="<xsl:value-of
                  select="."/>"</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 8 - Constructing the result tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result-tree nodes

Result-tree nodes are used both in the result tree and in the transformation

- the creation of the result tree
- T1 creating a result-tree fragment
- T2 creating a temporary tree

Recall the earlier processing model diagram (page 137)

- the diagram depicts the copying of nodes from the operation tree and the source tree to the result tree
  - the operation tree nodes that are copied to the result tree are the literal result elements
- also possible to explicitly add nodes of different types to the result tree

XSLT supports:

- direct construction of result tree nodes
- different ways to copy nodes from the source tree to the result tree
- constructing text nodes in the result tree reflecting numbering information found in the source tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing result-tree nodes (cont.)

The XSLT instructions covered in this chapter are as follows.

Instructions related to building the result tree:

- `<xsl:attribute>`
  - instantiate an attribute node in the result tree
- `<xsl:attribute-set>`
  - declare a set of attribute nodes for use in the result tree
- `<xsl:comment>`
  - instantiate a comment node in the result tree
- T2 `<xsl:document>`
  - instantiate a document node in the result tree
- `<xsl:element>`
  - instantiate an element node in the result tree
- T2 `<xsl:namespace>`
  - instantiate a namespace node in the result tree
- `<xsl:processing-instruction>`
  - instantiate a processing instruction node in the result tree
- `<xsl:text>`
  - instantiate a text node in the result tree
- `<xsl:copy>`
  - instantiate a copy of the current node in the result tree
- `<xsl:copy-of>`
  - instantiate a complete copy of a specified node in the result tree
- `<xsl:number>`
  - add a string to the result tree representing the position of the current node

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Building result tree nodes directly

XSLT has instructions to create the following kinds of result tree nodes:

- attribute
- T2 document
- element
- comment
- T2 namespace
- processing instruction
- text

P2 Validation against user-defined data types available when creating result tree nodes

- only when using a schema-aware processor
- only when turning on schema awareness

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing attribute nodes

Adding a single attribute

```
<xsl:attribute name="attribute-qname-AVT"
               namespace="optional-namespace-URI-for-qname-AVT">
  attribute content
</xsl:attribute>
```

- `name=` and `namespace=` are attribute value templates

T2 Alternative forms in XSLT 2.0

```
<xsl:attribute name="attribute-qname-AVT"
               namespace="optional-namespace-URI-for-qname-AVT"
               validation="declared-type-style"
               type="strict-using-named-type">
    attribute content
 </xsl:attribute>
```

- `validation=` and `type=` are mutually exclusive and optional
  - see Validating result tree nodes (page 186) for details on the attribute usage

```
<xsl:attribute name="attribute-qname-AVT"
               namespace="optional-namespace-URI-for-qname-AVT"
               validation="declared-type-style"
               type="strict-using-named-type"
               select="expression" separator="string-AVT"/>
```

Of note:

- add given attribute node to the result node tree for the current element
  - name includes namespace prefix, if desired
- specifying an invalid name for a node is an error
- the `select=` and `separator=` are the same as for `<xsl:value-of>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing attribute nodes (cont.)

Declaring a set of attributes

- only allowed as a top level instruction
- very useful for creating result trees with the XSL vocabulary due to the plethora of attributes that can be specified

```
<xsl:attribute-set name="attribute-set-qname">
  optional <xsl:attribute> instructions
</xsl:attribute-set>
```

- declare a number of attributes and their values with a name using `name=`

```
<xsl:attribute-set name="attribute-set-qname"
                   use-attribute-sets="attribute-set-qnames">
  optional <xsl:attribute> instructions
</xsl:attribute-set>
```

- declare a set of attributes from other sets of attributes using `use-attribute-sets=`

Using a set of attributes

```
<l-r-element xsl:use-attribute-sets="attribute-set-qnames"/>
```

- add a set or sets of attributes to the result node tree for the literal result element
- also available when copying nodes
- `xsl:use-attribute-sets=` a literal result element, `use-attribute-sets=` for some instructions
  - sets are useful for XSL formatting transformations where numerous attributes need to be specified

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing attribute nodes (cont.)

Constructing attributes:

- attached only to an element node of the result node tree
- before any of the given element's content is added to the result tree
- only way to selectively add an attribute to the result tree

Consider the need to produce:

```
<list intro="no" compact="yes" ordered="yes">
```

Many chances to define and redefine n attribute of a given name

- prioritized handling ensures predictable behavior
- `xsl:use-attribute-sets=` processed first
- attribute specified in a literal result element are added next
- executed `<xsl:attribute>` instructions are added last
- latter attribute specifications replace former for given element
- nothing can change for attributes once element content has been added

```
<xsl:attribute-set name="list-stuff">
  <xsl:attribute name="intro">yes</xsl:attribute>
  <xsl:attribute name="compact">yes</xsl:attribute>
</xsl:attribute-set>
...
<list intro="no" xsl:use-attribute-sets="list-stuff">
  <xsl:if test="@type='ol'">
    <xsl:attribute name="ordered">yes</xsl:attribute>
  </xsl:if>
  ...
</list>
```

- `name=` and `namespace=` are attribute value templates
- can use `<xsl:text>` to specify text content of an attribute
  - even though a text node is not added to the result tree
  - an attribute node is a leaf of the tree and never has children

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Constructing element nodes

Adding element nodes :

```
<xsl:element name="element-type-qname-AVT"
             namespace="optional-namespace-URI-for-qname-AVT"
             inherit-namespaces="yes-or-no-default-yes"
             validation="declared-type-style"
             type="strict-using-named-type"
             use-attribute-sets="optional-sets-of-attributes">
   optional-attribute-instructions
   optional-element-content
</xsl:element>
```

- add given element node to result tree
  - name includes namespace prefix, if desired
- specifying an invalid name for a node is an error
- `name=` and `namespace=` are attribute value templates
- `inherit-namespaces=` will attach the created element's namespace nodes to its child elements copied from the source tree
  - recall the process diagram on page 137
  - consider element `3'` is created with an instruction rather than being copied from element `3`
  - in such a case the namespace nodes inherited by `3'` from `A'` are not added to the copied nodes `4'` and `5'`
- `validation=` and `type=` are mutually exclusive and optional
  - see Validating result tree nodes (page 186) for details on the attribute usage
- sets of attributes are added using `use-attribute-sets=`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing element nodes (cont.)

Naming an element node with an attribute value template:

```
  <xsl:element name="H{count(ancestor-or-self::section) + 1}">
    ...template...
  </xsl:element>

  <xsl:element name="{local-name(.)}"><!--strip namespace prefix-->
    ...template...
  </xsl:element>
```

Provides protection against nuances regarding attached namespace nodes

- copying a literal result element node from the operation tree to the result tree will copy any attached namespace nodes
  - except those pruned by `exclude-result-prefixes=` attribute
- copying an element node using `<xsl:copy>` from the source tree to the result will copy any attached namespace nodes even if not used by the element node
- the `<xsl:element>` instruction synthesizes an element node without any unnecessary namespace baggage

Recall Namespaces (page 30)

- the local name is the name without any prefix

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing annotation nodes

Adding processing instruction and comment nodes :

```
<xsl:processing-instruction name="PI-target-AVT">
  processing-instruction-sequence-constructor
</xsl:processing-instruction>
```

```
<xsl:comment>
  comment-sequence-constructor
</xsl:comment>
```

T2 Alternative forms in XSLT 2.0:

```
<xsl:processing-instruction name="PI-target-AVT" select="expression"/>
```

```
<xsl:comment select="expression"/>
```

- specifying an invalid name for a node is an error
- `name=` is an attribute value template
- these nodes are leaves of the tree and never have children
- can use `<xsl:text>` to specify text content
- T2 the processor automatically injects a space after the first character of a sequence that matches the end delimiter
  - the sequence "`?>`" found in processing instruction text
  - the sequence "`--`" found in comment text or when the text ends with "`-`"
- T1 error condition when text content contains same sequence as the lexical delimiter for the construct
  - processing instruction text cannot contain "`?>`"
  - comment text cannot contain "`--`" or end with "`-`"
  - processor may choose not to report the error
    - will inject a space between the characters if no error reported

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing annotation nodes (cont.)

Contextual diagnostic information can be injected in XML comments in the result

- the diagnostic reporting by the processor typically emits all of the diagnostic messages as a stream without contextual information of when the messages are generated
- very helpful to synthesize an XML comment in the result tree with diagnostic information
- e.g. embedding a distinctive search string
  - `<!--debug: diagnostic information-->`
  - so as to make it easier to find the diagnostic information in the result
- e.g. embedding the XPath address of a source tree node
  - `<!--at: /doc/chapter[3]/fig[2]-->`
  - walking down the ancestor axis building the XPath address
  - some XML editing tools accept such an XPath address as a locator value
- e.g. embedding referential information for otherwise-cryptic links
  - `<xref idref="d1e27"/><!--Chapter 3: Repair tasks-->`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Constructing namespace nodes

Adding a namespace node :

```
<xsl:namespace name="namespace-prefix-AVT">
  namespace-URI-sequence-constructor
</xsl:namespace>

<xsl:namespace name="namespace-prefix-AVT"
               select="expression"/>
```

- `name=` is an attribute value template
- the namespace name (which is typically an absolute URI) is specified in either the `select=` attribute or the sequence constructor, but not both
  - the namespace name cannot be evaluated to the empty string
- an element must have been added to the result tree without yet any child content (but possibly with attached attributes and other namespaces) when this instruction is executed

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing document nodes

Creating a document node :

```
<xsl:document validation="declared-type-style"
              type="strict-using-named-type">
  document-sequence-constructor
</xsl:document>
```

- `validation=` and `type=` are mutually exclusive and optional
  - see Validating result tree nodes (page 186) for details on the attribute usage
- use this instruction to create a temporary tree variable or to constrain the result tree
- this instruction is *not* used as a replacement for `<xsl:result-document>`
  - that is used to create a serialized result tree
- consider that a temporary tree of a text node could be written as:

  ```
  <xsl:variable name="test">abc</xsl:variable>
  ```

  this is *not* a string and when formally declared must be expressed as a temporary tree as follows:

  ```
  <xsl:variable name="test" as="document-node()">
    <xsl:document>abc</xsl:document>
  </xsl:variable>
  ```
- depicted for variables in Variable and parameter binding (page 223)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing document nodes (cont.)

Also used for validating the result with an imported schema

- example with the result tree to validate the entire result

```
<xsl:template match="/">
  <xsl:document validation="strict">
    <xsl:apply-templates/>
  </xsl:document>
</xsl:template>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Constructing text nodes

- the expression is mandatory but the expression can calculate an empty string
- all items addressed are converted to strings (even text nodes) and the strings are separated with a space

Adding text nodes :

```
<xsl:text>escaped-text</xsl:text>

<xsl:text disable-output-escaping="yes">verbatim-text</xsl:text>
```

- `<xsl:text>` cannot have any sub-elements of any kind
- the XSLT processor is not obliged to respect `disable-output-escaping="yes"`
- `disable-output-escaping="yes"` can only be used for the text content of an element being added to the result tree
  - e.g. cannot be used within a result tree fragment variable assignment, or `<xsl:message>` construct
- use case is for non-XML/non-HTML markup (e.g. ASP code)
- to emit HTML entities, declare and use XML entities as shown on page 215

Stylesheet tree no-operation instruction:

```
 <xsl:for-each select="item">        <!--with new lines-->
   (<xsl:value-of select="."/>)
 </xsl:for-each>

 <xsl:for-each select="item">        <!--without new lines-->
   <xsl:text>(</xsl:text>
   <xsl:value-of select="."/>
   <xsl:text>)</xsl:text>
 </xsl:for-each>

 <xsl:for-each select="item">        <!--without new lines-->
   <xsl:text/>(<xsl:value-of select="."/>)<xsl:text/>
 </xsl:for-each>
```

- note above the use the empty text instruction to shape the non-white-space-only text nodes of the stylesheet tree
- the instruction `<xsl:text/>` adds nothing to the result tree but impacts on the text nodes of the stylesheet tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Escaping text placed in the result tree

XML and HTML serialization ensures well-formed output

- characters sensitive to markup processing are protected
- processor can choose any method it wishes to ensure characters are escaped
  - 1 using a CDATA section `<![CDATA[ ... ]]>`
    - for processors other than XSLT 1.0, CDATA sections cannot be used except as specified by the user
  - using a built-in entity reference under processor (not transform) control
    - any "<", "&" and ">" found in a text node can be emitted as `&lt;`, `&amp;` and `&gt;` respectively
    - the HTML output method may recognize HTML built-in entity references for non-ASCII characters
  - using a numeric character reference under processor (not transform) control
    - '<' can be serialized as '`&#60;`' or '`&#x3c;`' or '`&#x3C;`'
    - '&' can be serialized as '`&#38;`' or '`&#x26;`'
    - '>' can be serialized as '`&#62;`' or '`&#x3e;`' or '`&#x3E;`'
    - characters outside the serialization encoding can be emitted in decimal or hex
  - using a built-in or numeric character reference for attribute encoding based on attribute delimiters under processor (not transform) control
    - '"' can be serialized as '`&quot;`' or '`&#34;`' or '`&#x22;`'
    - "'" can be serialized as '`&apos;`' or '`&#39;`' or '`&#x27;`'
    - characters outside the serialization encoding can be emitted in decimal or hex

Stylesheets can only request the use of CDATA sections:

- `<xsl:output cdata-section-elements="`*`white-space-separated-qnames`*`"/>`
- uses a single CDATA section for each element in most cases
  - the sequence "]]>" in a text node, when be emitted in CDATA in XML, is as follows to ensure the CDATA is not prematurely terminated:
    - `<![CDATA[]]]><![CDATA[>]]>`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Escaping text placed in the result tree (cont.)

2 Using character maps is a declarative way of producing character representations

- see Character maps (page 199) for details

As a last resort can request no escaping

- if it is absolutely necessary to actually emit "<" or "&" from the result tree, the desire to do so can be indicated as follows:
- `<xsl:text disable-output-escaping="yes">`*`text`*`</xsl:text>`
- `<xsl:value-of disable-output-escaping="yes" select="`*`expr`*`"/>`
- compare with Serializing result text (page 153)
- it is very easy to produce XML that is not well-formed, or HTML that makes references to entities that do not exist; avoiding this technique ensures all output that is produced is well formed
- note that a conforming XSLT processor is not required to support the disabling of output escaping, so this technique is not guaranteed to be portable across all implementations
- can only be used for text generated for an element's content
  - not for content of `<xsl:attribute>`, `<xsl:comment>`, or `<xsl:processing-instruction>` instructions

Use case is only to put out sensitive markup characters not already produced by XSLT

- *not* to be used for emitting start and end tag syntax
  - XSLT produces a node tree and all tags are emitted from node processing
- *not* to be used for emitting named character entities such as ` `
  - to emit HTML entities, declare and use XML entities as shown on page 215
  - XSLT processors recognize valid Unicode characters and are required to emit these using any well-formed syntax it chooses

An example use-case where this is required is the emission of ASP code embedded in an HTML file:

```
01  <xsl:text disable-output-escaping="yes">
02            &lt;% Method-Call() %></xsl:text>
03  <xsl:text disable-output-escaping="yes"><![CDATA[
04            <% Method-Call() %>]]></xsl:text>
```

Both of the above instructions produce the following result (provided the attribute is respected by the processor):

```
01            <% Method-Call() %>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Escaping text placed in the result tree (cont.)

For example, the following instruction is often found in stylesheets because it can be used to coerce the XSLT processor to emit "` `" in order to reference the entity definition built-in to HTML, but it only works when serializing the result tree into markup syntax:

```
01 <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
```

- this technique should not be used because the result tree does not include an entity reference, just non-escaped text
- the two ways to accomplish this in a portable fashion supported by all conforming XSLT processors that support the HTML output method are:
  - to define the entity `nbsp` as ` ` and reference the entity in the stylesheet
  - to reference the character entity ` ` directly in the stylesheet
  - see Internal general entities (page 215)
- in both the above cases, the character is seen to be a non-ASCII character and is recognized by the HTML output method as a known built-in entity, and emitted as the entity reference ` `

Remember that the input source file still always has to be well-formed XML, hence the stylesheet must escape the characters in the input that are not going to be escaped in the output by:

- using built-in or declared character entity references
- using numeric character references
- using CDATA sections

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Escaping text placed in the result tree (cont.)

Consider the need to produce the following result:

```
01 <html>
02 <body>
03 <p>This is an nbsp reference: ' '.</p>
04 </body>
05 </html>
```

The XSLT script `disable.xsl` disables the output escaping method to emit "` `":

```
01 <?xml version="1.0"?><!--disable.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04                 version="1.0">
05
06 <xsl:output method="html" indent="yes"/>
07
08 <xsl:template match="/">                    <!--root rule-->
09  <html>
10   <body>
11    <p>
12     <xsl:text>This is an nbsp reference: '</xsl:text>
13      <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
14      <xsl:text>'.</xsl:text>
15    </p>
16   </body>
17  </html>
18 </xsl:template>
19
20 </xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Escaping text placed in the result tree (cont.)

The XSLT script `disable-not.xsl` approaches the same problem without circumventing the protection available in XSLT:

```
<?xml version="1.0"?><!--disable-not.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nbsp " ">   <!--declare HTML value of the entity-->
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

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

<xsl:template match="/">                          <!--root rule-->
 <html>
  <body>
   <p>
    <xsl:text>This is an nbsp reference: ' '.</xsl:text>
   </p>
  </body>
 </html>
</xsl:template>

</xsl:stylesheet>
```

Note how the identical output is produced by using an entity reference in the above stylesheet:

- the reference points to the entity declaration in the stylesheet
  - this could have been included as one of a set of declarations for the entire set utilized by HTML
- the XML processor in the XSLT processor translates the reference into the entity content and only the character ` ` is stored in the stylesheet tree (not the entity reference)
- the XSLT script adds the character ` ` to the result tree
- the HTML output method recognizes the ` ` character as matching that value for the ` ` entity and emits the entity reference instead
  - defined *only* for the HTML output method
  - not recognized by other output methods

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Escaping text placed in the result tree (cont.)

Chapter 8 - Constructing the result tree
Section 1 - Result tree node instantiation

It may still be necessary to take advantage of this approach:

- to solve problems not solved in a well-behaved fashion for the current implementation of XSLT
- for a processor that implements output escaping while not supporting emitting the document type declaration
- to deliver packages of non-well-formed markup from a server process to a browser parsing the output information as markup (e.g. from a database of HTML fragments)

During the processing of the root node (typically, but not necessarily), a block of non-escaped text can be emitted to produce the document type declaration, as in the following example:

- version 1.0 of XML doesn't support the definition of the output internal declaration subset of the instance
- the correct method would be to use `doctype-public=` or `doctype-system=`, but these may not be supported by the XSLT processor
  - if the internal declaration subset is needed, one cannot use these methods since the declaration is ended in order to correctly emit the document element

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Escaping text placed in the result tree (cont.)

Consider the following stylesheet to emit a complete document type declaration by disabling the output escaping on a text node in which the entire prologue is written using a CDATA section to reduce the amount of per-character escaping required in the stylesheet:

```
01 <?xml version="1.0"?><!--disable-decl.xsl-->
02 <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03 <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04                 version="1.0">
05
06 <xsl:output method="xml" indent="yes" omit-xml-declaration="yes"/>
07
08 <xsl:template match="/">                    <!--root rule-->
09  <xsl:text disable-output-escaping="yes"><![CDATA[
10 <!DOCTYPE test [
11 <!ENTITY nbsp " ">
12 ]>]]></xsl:text>
13  <test>
14   <xsl:text>This is an nbsp reference: '</xsl:text>
15   <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
16   <xsl:text>'.</xsl:text>
17  </test>
18 </xsl:template>
19
20 </xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree

Copying arbitrary nodes and all of their descendants (deep copy):

```
01 <xsl:copy-of select="result-tree-fragment-or-temp-tree-expression"/>
```

- fragment is added to the result tree verbatim

```
01 <xsl:copy-of select="node-set-expression"/>
```

- nodes from source tree are added to the result tree in document order
- 1 2 nodes from a temporary tree are added to the result tree in document order
- template rules are not triggered (this is not a "push")

```
01 <xsl:copy-of select="other-expression"/>
```

- string value is added as in <xsl:value-of>

Copying the current node (shallow copy):

```
01 <xsl:copy>
02   optional-sequence-constructor
03 </xsl:copy>
```

- copies the current node (type and, if named, name) to the result node tree
- the type or name of the source node need not be known
- sub-elements and attributes from the source node tree are *not* automatically copied
  - any content copying must be explicitly requested in the stylesheet

```
01 <xsl:copy use-attribute-sets="white-space-separated-qnames">
02   optional-sequence-constructor
03 </xsl:copy>
```

- copies the current node and adds sets of attributes along the way

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree (cont.)

- Additional features available in XSLT 2.0 for `<xsl:copy-of/>` and `<xsl:copy>`
  - `validation="`*`declared-type-style`*`"` or `type="`*`strict-using-named-type`*`"`
    - using one of these mutually-exclusive attributes validates the copied node against either a global schema declaration or a specified type
    - style values are "`strict`", "`lax`", "`preserve`" and "`strip`"
      - see Validating result tree nodes (page 186) for details
  - `copy-namespaces="`*`yes-or-no-default-yes`*`"`
    - default value is "`yes`" and applies only to elements
    - the value "`no`" will prevent unneeded namespace nodes attached to the source element from being copied to the target element
    - illustrated on page 137
      - in the diagram this is the copying of unused namespace nodes from node `3` when creating node `3'`
  - `inherit-namespaces="`*`yes-or-no-default-yes`*`"`
    - for `<xsl:copy>` and literal result elements only
    - e.g. `<`*`l-r-e`*` xsl:inherit-namespaces="no">`*`...nodes...`*`</`*`l-r-e`*`>`
    - applies only to the future children in the result tree of the element being added
    - the value "`yes`" will attach the target element's namespace nodes to its element children
    - descendents will need namespace declarations for all namespace nodes not attached or found in the ancestry
    - may trigger `xmlns=""` (or `xmlns:`*`prefix`*`=""` for XML 1.1) in child elements
    - illustrated on page 137
      - in the diagram this is the copying of unused namespace nodes from node `A` when creating node `3'`
    - inheriting nodes from the operation tree happens after copying nodes from the source tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree (cont.)

Sample source data and stylesheet illustrating the processing model on page 137:

```
<sns1:s1 xmlns:sns1="sns1" xmlns:sns2="sns2" xmlns="sdef">
  <sns1:s2/>
  <sns1:s3>
    <sns1:s4 ons1:new1="z" xmlns:ons1="ons1"/>
    <sns1:s5 sns2:new2="y"/>
  </sns1:s3>
</sns1:s1>
```

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:sns1="sns1" exclude-result-prefixes="sns1"
                version="2.0">

<xsl:output indent="yes" method="xml" version="1.0"/>

<xsl:template match="/">
<a xmlns="idef" xmlns:ons1="ons1" xsl:inherit-namespaces="yes">
  <b/>
   <xsl:copy-of select="doc('inherit-namespaces.xml')//sns1:s3"
                copy-namespaces="yes"/>
  <d/>
</a>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree (cont.)

Resulting combinations of copying and inheriting namespaces on page 137:

```
Not copying namespaces, not inheriting namespaces
<a xmlns="idef" xmlns:ons1="ons1">
   <b/>
   <sns1:s3 xmlns:sns1="sns1" xmlns="">
      <sns1:s4 xmlns:ons1="ons1" ons1:new1="z"/>
      <sns1:s5 xmlns:sns2="sns2" sns2:new2="y"/>
  ...
Not copying namespaces, inheriting namespaces
<?xml version="1.0" encoding="UTF-8"?>
<a xmlns="idef" xmlns:ons1="ons1">
   <b/>
   <sns1:s3 xmlns:sns1="sns1">
      <sns1:s4 ons1:new1="z"/>
      <sns1:s5 xmlns:sns2="sns2" sns2:new2="y"/>
  ...
Copying namespaces, not inheriting namespaces
<?xml version="1.0" encoding="UTF-8"?>
<a xmlns="idef" xmlns:ons1="ons1">
   <b/>
   <sns1:s3 xmlns:sns1="sns1" xmlns:sns2="sns2" xmlns="sdef">
      <sns1:s4 xmlns:ons1="ons1" ons1:new1="z"/>
      <sns1:s5 sns2:new2="y"/>
  ...
Copying namespaces, inheriting namespaces
<?xml version="1.0" encoding="UTF-8"?>
<a xmlns="idef" xmlns:ons1="ons1">
   <b/>
   <sns1:s3 xmlns:sns1="sns1" xmlns:sns2="sns2" xmlns="sdef">
      <sns1:s4 ons1:new1="z"/>
      <sns1:s5 sns2:new2="y"/>
  ...
```

- note the need for the redundant `ons1` namespace declaration when not inheriting namespace nodes
- note the need for the `sns2` namespace on `s5'` when not copied onto `s3'`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree (cont.)

Examples contrasting the copying of nodes

An example code fragment to copy only the given element and its attributes, but to process its content without blindly copying the content, is as follows:

```
<xsl:template match="this|that|other">
  <xsl:copy>            <!--copy whatever the matched node is-->
    <xsl:copy-of select="@*"/>   <!--copy attached attr nodes-->
    <xsl:apply-templates/>              <!--push child nodes-->
  </xsl:copy>
</xsl:template>
```

An example code fragment to copy the entire given element and all its content:

```
<xsl:template match="this|that|other">
  <xsl:copy-of select="."/>   <!--copy the entire current node-->
</xsl:template>
```

Contrast this to the execution of `<xsl:value-of select=".">` on a node of the source node tree that adds to the result tree the concatenation of descendent text nodes as a string of text.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Copying source tree nodes to the result tree (cont.)

Diagnostic stylesheet example

```
<?xml version="1.0"?><!--copyofall.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">          <!--entire copy of root node-->
  <xsl:copy-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

- `copyofall.xsl` will copy the entire input XML to the output:
  - recall the physical entity structure of the input in Extensible Markup Language (XML) (page 9)
  - the entire physical structure of the stylesheet is written into the output as a single file
    - all external XML markup fragments consolidated into a single file
    - imported and included files are *not* consolidated using this technique
  - all general entities are resolved, thereby revealing what the XSLT processor is seeing as the value of entities that may be parameterized through marked sections

Modification for "pretty-print" of an XML file:

- `<xsl:output indent="yes"/>`
- when an XML file is not indented it can be very difficult to find information
  - this addition to the stylesheet serializes the result tree indented
  - helps to find errors that are listed by line number when there are concatenated lines in the file

Useful for a performance diagnostic

- timing this stylesheet on data determines the cost of building the input tree and serializing the result tree
- comparing the timing of this stylesheet on your data with the timing of your stylesheet on your data reveals the approximate amount of time being spent in your stylesheet
- comparing two versions of your stylesheet for performance should be comparing the differences between each stylesheet and the base cost of tree building and serialization

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Copying source tree nodes to the result tree (cont.)

Chapter 8 - Constructing the result tree
Section 2 - Copying nodes

Identity transform with modifications

Very common to need to make only very small changes to an XML document

- an identity transform will reconstruct in the result tree all of an XML source tree in a piecemeal fashion
- matching template rules for specific nodes allow one to override the copy

In the following `identity.xsl` example, all `<c>` elements are changed and `e=` attributes are removed:

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

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

<!--specializations-->

<xsl:template match="a">  <!--change the a element; ignore attrs-->
  <new-A-here>
    <xsl:apply-templates/>
  </new-A-here>
</xsl:template>

<xsl:template match="@b"/> <!--remove b attributes-->

</xsl:stylesheet>
```

A more general identity template:

- typically used in a larger stylesheet with more stylesheet logic
- note how parameters are not passed with this template, as with the built-in templates
  - using tunnel parameters will ensure passed values are accessible to called templates

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


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Copying source tree nodes to the result tree (cont.)

Acting on this data:

```
<test>
  <a a="test">a content</a>
  <b b="test">b content</b>
  <c c="test">c content</c>
</test>
```

The example complete stylesheet produces this result where only the specialized template rules have made a change:

```
<?xml version="1.0" encoding="utf-8"?><test>
  <new-A-here>a content</new-A-here>
  <b>b content</b>
  <c c="test">c content</c>
</test>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Copying source tree nodes to the result tree (cont.)

Chapter 8 - Constructing the result tree
Section 2 - Copying nodes

Tweaking importable stylesheet example

- `copyall.xsl` will copy the entire input XML to the output providing for customization:
  - when imported into another stylesheet, the importing stylesheet can specialize the behavior for specific node template rules
  - where it is necessary to copy *most* of a file but change only a few items, this script can be a helpful shell within which to write the customizations

```
<?xml version="1.0"?><!--copyall.xsl-->
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:copyall="http://www.CraneSoftwrights.com/ns/copyall"
   exclude-result-prefixes="copyall"
   version="1.0">

<!--use the following to copy all of the content of a node-->

<xsl:template name="copyall:copy-content">
  <xsl:apply-templates
      select="@*|processing-instruction()|comment()|*|text()"/>
</xsl:template>

<!--default behaviours for all nodes-->

<xsl:template match="/|
                  processing-instruction()|comment()|*|@*|text()">
  <xsl:copy>
    <xsl:call-template name="copyall:copy-content"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Copying source tree nodes to the result tree (cont.)

Importing the `copyall.xsl` stylesheet on requires the declaration and exclusion of the stylesheet's namespace:

```
<?xml version="1.0"?><!--copyalltest.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training-->
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:copystuff="http://www.CraneSoftwrights.com/ns/copyall"
   exclude-result-prefixes="copystuff"
   version="1.0">

<xsl:import href="copyall.xsl"/>

<xsl:template match="b">                <!--rename an element -->
  <new-b>
    <xsl:call-template name="copystuff:copy-content"/>
  </new-b>
</xsl:template>

<xsl:template match="d">              <!--redefine an element -->
  <new-d>New d-value here</new-d>
</xsl:template>

<xsl:template match="@a">             <!--rename an attribute -->
  <xsl:attribute name="new-a">
    <xsl:value-of select="."/>     <!--this copies the content-->
  </xsl:attribute>
</xsl:template>

<xsl:template match="@c">           <!--redefine an attribute -->
  <xsl:attribute name="{name(.)}">
    <xsl:text>new-c</xsl:text>
  </xsl:attribute>
</xsl:template>

<xsl:template match="f|@g"/><!--remove an element or attribute-->

</xsl:stylesheet>
```

Note that it is only a convention that the namespace URI also be an addressable

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Copying source tree nodes to the result tree (cont.)

Chapter 8 - Constructing the result tree
Section 2 - Copying nodes

Consider the following input instance with elements and attributes:

```
<?xml version="1.0"?>
<test>
  <a a="test">a content</a>
  <b b="test">b content</b>
  <c c="test">c content</c>
  <d d="test">d content</d>
  <e e="test">e content</e>
  <f f="test">f content</f>
  <g g="test">g content</g>
</test>
```

The stylesheet above will produce the following output instance:

```
<?xml version="1.0" encoding="utf-8"?>
<test>
  <a new-a="test">a content</a>
  <new-b b="test">b content</new-b>
  <c c="new-c">c content</c>
  <new-d>New d-value here</new-d>
  <e e="test">e content</e>

  <g>g content</g>
</test>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Building result tree nodes with literal result elements

Recall that elements in the stylesheet that are in templates and are not XSLT instructions are called "literal result elements":

- when a literal result element is added to the result tree, it is copied as an element node
- any specified attributes are instantiated in the result tree after being interpreted as attribute value templates

Literal result element XSLT attributes:

- allowed in addition to any result vocabulary attributes specified in the literal result element
- where scoped, influence only descendants of the given literal result element

`xsl:extension-element-prefixes="`*`white-space-separated-prefixes`*`"`

- to declare the namespace prefixes of element instructions available from the XSLT processor (see The stylesheet document/container element (page 179))
- without which element from such prefixes are considered to be literal result elements and not instructions

`xsl:exclude-result-prefixes="`*`white-space-separated-prefixes`*`"`

- to inhibit the contingent creation of namespace nodes (see The stylesheet document/container element (page 179))
- does not inhibit the required creation of namespace nodes

`xsl:version="`*`numeric-version`*`"`

- to declare the specification level of XSLT required by the stylesheet instructions (see The stylesheet document/container element (page 176))

`xsl:use-attribute-sets="`*`white-space-separated-qnames`*`"`

- to add to the result element all attributes in a previously declared collection of attributes

`xsl:inherit-namespaces="`*`yes-or-no-default-yes`*`"`

- the attached namespace nodes of the literal result element are not automatically attached to the element's element children

Should not use the default namespace for the XSLT instruction namespace

- prevents the above attributes from being used in literal result elements
- attributes without namespace prefixes are in no namespace, not in the default namespace

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering

Adding a text node whose string value represents numbers to the result tree

- calculate the numeric result of an expression and render the text representation
  - rounded to an whole number value
- reveal the ordinal position of a source node in various source tree contexts as a number
- behavior is based on the presence of the `value=` attribute

Non-source-tree-oriented when using `value=`:

- the context is the current node list
  - the *only* time the context for `<xsl:number/>` is the current node list
- useful when `<xsl:value-of select="`*`expression`*`">` rendering as simple floating point numbers is insufficient
- `<xsl:number value="`*`arbitrary-expression`*`" format="`*`token-AVT`*`"/>`
  - render the calculated rounded whole number value of the expression
- `<xsl:number value="position()" format="i"/>`
  - the position of the current node within the current node list
  - formatted in lower-case roman numerals

Typical source-tree-oriented use when not using `value=`:

- processor-calculated "identification of quantity"
  - declarative invocation supplants the need to algorithmically implement counting in the stylesheet
- the context is the current node in the source node tree
- `<xsl:number/>`
  - the position of the current node within like-named source tree sibling nodes

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering (cont.)

Source tree context attributes available:

- all attributes are evaluated as attribute value templates
- `select="`*`singleton-node-expr`*`"`, `count="`*`union-match-pattern`*`"`, `from="`*`union-match-pattern`*`"`, and `level="`*`type`*`"` characterize the numbering

1 2 `select="`*`singleton-node-expr`*`"`

- repositions the current node to specify the context

`count="`*`union-match-pattern`*`"`

- what nodes are being counted
  - may be a simple pattern such as `count="para"`
  - the current node itself or the first ancestral node that matches the pattern determines which siblings are used
- union operator may be used
  - to count nodes of different types or names in the calculation
  - `count="para|fig|list"` indicates all element nodes named either "`para`", "`fig`" or "`list`" are to be considered in the calculation
- may be omitted for current node
  - indicates only the type and (if named) name of the current node

`from="`*`union-match-pattern`*`"`

- where in the hierarchy to count nodes from
  - when counting amongst siblings, this pattern represents descendent nodes of a node on the `ancestor::` axis
  - when counting across sibling boundaries, represents descendent and following nodes of a node on the `preceding::` and `ancestor-or-self::` axes
- specifies the point *below* which counting begins
- may be omitted for the entire document
  - indicates descendants of the root node (all nodes)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Source tree numbering (cont.)

Chapter 8 - Constructing the result tree
Section 3 - Numbering instructions

Numbering attributes (cont.):

`level="single"`

- a single number counted amongst preceding siblings
  - returns a single number of the count of the counted nodes
  - starting at the closest ancestral node that is a descendant of the node matching the `from="`*`pattern`*`"` pattern
  - amongst that node's `preceding-sibling::` and `self::` axes
- for example, numbering paragraphs when processing a paragraph:
  - `<xsl:number/>`
    - only the current node type is included in the count
  - `<xsl:number count="para|fig" level="single"/>`
    - both paragraphs and figures are included in the count

`level="any"`

- a single number counted amongst previous elements
  - returns a single number of the count of all counted nodes
  - amongst the current node's `preceding::` and `ancestor-or-self::` axes
  - starting after the closest node matching `from="`*`pattern`*`"`
- for example, numbering figures when processing a figure:
  - `<xsl:number level="any"/>`
    - document-scope figure number
  - `<xsl:number level="any" from="sect"/>`
    - section-scope figure number

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Source tree numbering (cont.)

`level="multiple"`

- a set of numbers (tumbler) counted at levels of ancestry
  - returns a set of numbers of the counts of the counted nodes amongst their siblings at different levels in the hierarchy
  - at each level of document hierarchy along the `ancestor-or-self::` axis up to the descendant of the closest ancestral node matching the `from="`*`pattern`*`"` pattern
  - amongst each level ancestor's `preceding::` and `ancestor-or-self::` axes
  - only at levels of ancestry where counted nodes exist (i.e. no tumbler value is ever zero)
- for example, a chapter, section, subsection tumbler:
  - `<xsl:number count="chap|sect|subs" level="multiple"/>`
    - Chapter "1", Section "1.1", Subsection "1.1.1", Section "1.2", Subsection "1.2.1", Chapter "2", Section "2.1", Subsection "2.1.1", Section "2.2", etc.
  - `<xsl:number count="sect|subs" from="chap" level="multiple"/>`
    - Chapter, Section "1", Subsection "1.1", Section "2", Subsection "2.1", Chapter, Section "1", Subsection "1.1", Section "2", etc.

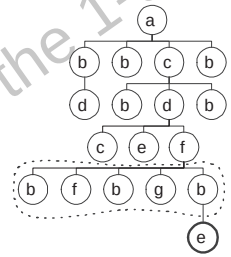
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering (cont.)

In each of the `<xsl:number/>` illustrations, the context node is the deepest element node named "`e`" (shown with a bolded border) and the values calculated differ only by the attributes used:

An illustration of `level="single"`:

```
01  <xsl:number level="single" count="b"/>
```



- the closest hierarchical element being counted, "`b`", is the parent of `e`
- only the found ancestor and its preceding siblings are considered in the calculation
  - note the scope is the combination of nodes on the `preceding-sibling::` and `self::` axes
- the text node "`3`" is added to the result tree
- without `count=`, only nodes of the current type and name would have been counted (in this example "`e`"), adding the text node "`1`" to the result tree

Example of use of `level="single"` to indicate the number of the `<module>` elements from the ancestral `<course>` element to the current location in the source node tree (current module number within the current course):

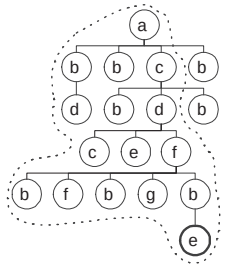
```
01  <xsl:text>&Module; </xsl:text>
02  <xsl:number level="single"
03              from="course"
04              count="module"/>
05  <xsl:text>: </xsl:text>
06  <xsl:value-of select="ancestor-or-self::module/title"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering (cont.)

An illustration of `level="any"`:

```
01  <xsl:number level="any" count="b"/>
```



- the text node "`6`" is added to the result tree
- note the scope is the combination of nodes on the "`preceding::`" and "`ancestor-or-self::`" axes after finding the closest ancestor
- without `count=`, only nodes of the current type and name would have been counted (in this example "e"), adding the text node "`2`"

Example of use of `level="any"` to indicate the number of `<figure>` elements from the start of the source node tree (the document-wide figure number):

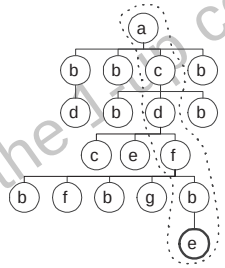
```
01  <xsl:text>&Figure; </xsl:text>
02  <xsl:number level="any"
03              count="figure"
04              format="i"/>
05  <xsl:text>: </xsl:text>
06  <xsl:value-of select="title"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering (cont.)

An illustration of `level="multiple"`:

```
01  <xsl:number level="multiple" count="b|c|d|g" format="1-1"/>
```



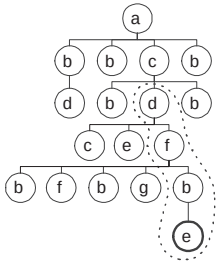
- the text node "`3-2-4`" is added to the result tree
- a tumbler is included in the text node for each level of the `ancestor-or-self::` hierarchy where there are any counted elements at that level (a level of hierarchy is skipped if that level has no members of the `count=` pattern)
- each element in the hierarchy is counted relative to its `preceding-sibling::` and `self::` axes
  - cousins are not included in the count
- when no `from=` is specified, the counting is from descendants of the root node
- all members of the `count=` pattern are counted at each level as siblings

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Source tree numbering (cont.)

Another illustration of `level="multiple"` adding `from=` and using the default formatting:

```
01  <xsl:number level="multiple" count="b|c|d|g" from="c"/>
```



- the text node "`2.4`" is added to the result tree
- note the hierarchy examined is along the "`ancestor-or-self::`" axis counting only from the descendants of the closest ancestral element node named with the `from=` attribute
- note the scope is the combination of nodes on the "`preceding::`" and "`ancestor-or-self::`" axes that are descendants of the closest ancestral element node named with the `from=` attribute
- the node being counted from is not included even if it is one of the node names being counted

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Formatting numbers as a sequence of characters

A token can be used to represent how numbers are to be rendered as a sequence of characters:

- for `<xsl:number/>` in XSLT and for page numbering in XSL-FO
- format can specify a prefix and terminator for all and a separator for "`multiple`"
- default format is "`1`" with a period separator (not terminator) for "`multiple`"

`format="`*`token-AVT`*"

- specifies the counting scheme to be used when formatting the value
- `format="1"` counts `1, 2, ..., 9, 10, 11, ..., 99, 100, 101, ...`
- `format="01"` counts `01, 02, ..., 09, 10,..., 99, 100, ...`
  - each "`0`" prefix is a zero-fill indication for number values formatted less than the length of the format string
  - `grouping-separator="`*`AVT`*" e.g. "`,`" specifies the character between digit groups
  - `grouping-size="`*`AVT`*" e.g. "`3`" specifies the number of digits in each group (e.g.: 1,000,000)
- `format="i"` counts `i, ii, iii, iv, v, ..., ix, x, xi, ...`
- `format="I"` counts `I, II, III, IV, V, ..., IX, X, XI, ...`
- `format="a"` counts `a, b, ..., z, aa, ab, ac, ...`
- `format="A"` counts `A, B, ... Z, AA, AB, AC, ...`
- `format="`*`a-Unicode-character`*" specifies a translation
  - converts the number into a representation based upon a specific language, pointing to the "zero" character with a Unicode digit class
  - `letter-value="`*`alphabetic-or-traditional-AVT`*" for ambiguous distinctions
    - distinguishes numbering schemes in those languages where the first character of the sequence is ambiguous
    - unlike English where the differing first characters of "`a`" and "`i`" distinguish alphabetic and roman numeral formats
- 1.2 `format="w"` counts `one, two, ...`
- 1.2 `format="W"` counts `ONE, TWO, ...`
- 1.2 `format="Ww"` counts `One, Two, ...`

1.2 `ordinal="`*`ord-AVT`*" changes cardinal numbers to ordinal numbers

- `ordinal="yes"` with `format="1"` produces "`1st`", "`2nd`", etc.
- `ordinal="yes"` with `format="w"` produces "`first`", "`second`", etc.
- `ordinal="yes"` with `format="W"` produces "`FIRST`", "`SECOND`", etc.
- `ordinal="yes"` with `format="Ww"` produces "`First`", "`Second`", etc.

`lang="`*`indication-AVT`*"

- indication of the language of words and letters (using same values as `xml:lang`)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Formatting numbers as a sequence of characters (cont.)

When using `level="multiple"` a number of tokens can be specified:

- each component (digits and separators) of the resulting string can be formatted differently
- the digits are formatted by corresponding format tokens
- the separators used between components are described by sequences appearing before format tokens
- the period "." is used as the separator sequence when no separator sequences are specified
- the last format token specified is repeated for unspecified format tokens
- the last separator sequence specified (that which is before the last format token) is repeated for unspecified separator tokens
- a termination sequence may be specified (that which is after the last format token) and is displayed after the last digits value

Consider the example where the number "1.4.2" needs to be formatted:

format attribute	resulting text node
absent	`1.4.2`
`format="1.a.i"`	`1.d.ii`
`format="1.a"`	`1.d.b`
`format="A.1"`	`A.4.2`
`format="i"`	`i.iv.ii`
`format="1:a-i"`	`1:d-ii`
`format="1:a"`	`1:d-b`
`format="1---a"`	`1---d---b`
`format="1---a***i"`	`1---d***ii`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Chapter 9 - Sorting and grouping

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Sorting and grouping

This chapter covers how to arrange the construction of results in ordered fashion.

Sorting

An important part of many transformations is the need to re-order the information in source tree nodes into a sorted order for processing into result tree nodes:

- designed for sorting the context list using multiple criteria
  - each criterion is a single value calculated for each node
  - value may be simple node value or may be any XPath evaluation relative to node
- the source tree is untouched during sorting
  - items being sorted are selected from the tree and the selection itself is sorted, not the tree
- sorting can be language based, numeric based, type-based or based on custom semantics
- the context list can be any arbitrary sequence
- multiple keys are used to sort clumps of equal values by other values
  - e.g. a secondary key is used when the primary key finds clumps of equal values
    - the secondary key is only applied to the clumped values, maintaining the set of clumped values in the same position of the primary sort
  - e.g. a tertiary key is used when the secondary key finds clumps of equal values
    - the tertiary key is only applied to the clumped values after the secondary sort, maintaining the set of clumped values in the same position of the secondary sort, in the same position of the primary sort
  - leftover clumped values after all keys are accommodated are left in context list sequence order

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sorting and grouping (cont.)

Grouping and uniqueness

Another important part of many transformations is the need to infer structure from the results of sorting information, which is a process often called "grouping":

- collecting information while separating and grouping it by common values
  - i.e. grouping the clumps under the value that created the clump
  - selecting a single piece of composite information obtains all components
  - a simple sort doesn't partition the composite information into constituent pieces
- specific application of the generalized problem of finding unique values from a set
  - often necessary to find unique values in a set of values
  - unique values make up the group headings
- 1.1 no explicit support for grouping under duplicate source tree node values
- 1.2 explicit support for grouping under duplicate source tree node values

1.1 This chapter covers three techniques of using XSLT 1.0 to group constructs when processing:

- using reverse-document-order axes
- the "Muenchian Method" of using `<xsl:key>`
- using variables

1.2 This chapter also covers the built-in XSLT 2.0 facility for grouping

- `group-adjacent`, `group-by`, `group-starting-with` and `group-ending-with`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sorting and grouping (cont.)

The XSLT instructions covered in this chapter are:

- `<xsl:sort>`
  - specify a criterion with which to sort a set of nodes
- 2 `<xsl:perform-sort>`
  - specify the criteria with which to sort a sequence of items
- 2 `<xsl:for-each-group>`
  - act on a set of items according to grouping criteria

The XSLT functions covered in this chapter are:

- 2 `current-grouping-key()`
  - returns the value by which members of the current set of grouped items are grouped
- 2 `current-group()`
  - returns the members of the current set of grouped items

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sorting opportunities

Sorting is performed on a context list created using `select=`

- rearranges the context list from sequence order into sorted order before acting on the template
- the original order is not available to the template processing
- remember that the document tree is read-only and cannot be changed
  - the sort never changes the document tree, only the context list

Sorting is available when processing templates:

- push items through the stylesheet:
- `<xsl:apply-templates select="`*`node-set-expression`*`">`
  `...multiple <xsl:sort> instructions...`
- pull items into the stylesheet:
- `<xsl:for-each select="`*`sequence-expression`*`">`
  `...multiple <xsl:sort> instructions...`
  `...template...`

2 Also available when sorting sequences for processing (see page 406):

- returning a re-ordered sequence of nodes or values:
- `<xsl:perform-sort select="`*`sequence-expression`*`">`
  `...multiple <xsl:sort> instructions...`
- returning an ordered sequence of generated nodes or values:
- `<xsl:perform-sort>`
  `...multiple <xsl:sort> instructions...`
  `...template and/or <xsl:sequence> constructors...`

2 Also available when grouping items for processing (see page 434):

- pulling the first member of each group:
- `<xsl:for-each-group select="`*`sequence-expression`*`">`
  `...multiple <xsl:sort> instructions...`
  `...template...`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sorting sequences

T2 Used return a sequence in sorted order

- e.g. in a variable declaration
- e.g. in a user function callable from XPath

The source of nodes is the `select=` or the body, but not both:

T2
```
<xsl:perform-sort select="sequence-expression">
  ...multiple <xsl:sort> instructions...
</xsl:perform-sort>
```

- `select="`*`sequence-expression`*`"` defines the initial sequence
- the result of the instruction is the sequence sorted according to the sort criteria

T2
```
<xsl:perform-sort>
  ...multiple <xsl:sort> instructions...
  ...template and/or <xsl:sequence> constructors...
</xsl:perform-sort>
```

- the sequence constructor defines the initial sequence
- use `<xsl:sequence select="`*`expr`*`">` to select values using XPath
- use content and constructors to create values using XSLT
- the result of the instruction is the sequence sorted according to the sort criteria

An example of a sorted variable of nodes:

- ```
  01 <xsl:variable name="sorted-nums" as="element(num)+">
  02   <xsl:perform-sort select="/nums/num">
  03     <xsl:sort data-type="number"/>
  04   </xsl:perform-sort>
  05 </xsl:variable>
  ```
- recall the importance of the `as=` from page 223
  - without the `as=` the variable would be a tree not a sequence

An example of a function used to sort a collection of arbitrary items:

- in support of a user function `my:sort(`*`sequence`*`)` for use in XPath
- ```
  01 <xsl:function name="my:sort" as="item()*">
  02   <xsl:param name="in" as="item()*"/>
  03   <xsl:perform-sort select="$in">
  04     <xsl:sort select="."/>
  05   </xsl:perform-sort>
  06 </xsl:function>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The sort instruction

`<xsl:sort select="`*`optional-expression`*`"/>`

- express a single sort criterion based on an expression value
- one instruction for each sort key
  - typically `select="`*`relative-node-set-expression`*`"`
    - evaluated relative to each node of the node set
    - when absent the processor uses value of each node being sorted
      - `select="."`
- multiple sort keys specified in sequence order
  - primary, secondary, tertiary, etc.
- must all appear before any template content
  - need not appear before `<xsl:with-param>`
- T1 equal sort values are sorted in sequence order
- T2 equal sort values are sorted in sequence order only if the sort is stable (default)

The sorted node set becomes the current node list

- defines XPath context of current node list and current node
- current node list is processed after being sorted
  - sorting does not rearrange the source node tree, only the current node list
- the `position()` function always reflects the order of the current node list
  - the order of the unsorted current node list before the sort
    - can be used in `select=` to invert the sequence order
  - the order of the sorted current node list after the sort
- references through the axes are *not* affected
  - nodes along axes are always regarded in proximity order in expressions
  - sets of nodes from axes are always regarded in document order

The `<xsl:sort>` instruction is not considered part of the template

- syntactically included within the template of the instruction
- is consumed by the instruction interpretation and ignored when processing individual nodes

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

The sort instruction (cont.)

Sort key evaluation criteria attributes:

- each evaluated as an attribute value template
- the sort order
  - `order="`*`ascending-or-descending-AVT`*`"`
  - default is "`ascending`"
- 2 `stable="`*`yes-or-no-AVT-default-yes`*`"`
  - only allowed on the first `<xsl:sort>` of a set of sort directives
  - a value of "`yes`" (default) returns sorted-equal items in sequence order
  - a value of "`no`" returns sorted-equal items in an implementation order
- the sort nature is specified using `data-type="`*`AVT`*`"`
  - `data-type="text"` (default basis in XSLT 1.0)
    - indicates the sort order is lexicographic based on the string text of the sort key values
    - 2 Unicode code point order default is different than XSLT 1.0 default
      - important when migrating stylesheets from 1.0 to 2.0
    - 2 `collation="`*`uri-string-AVT`*`"` (default for text data type in XSLT 2.0)
      - overrides Unicode collation (see page 289 for details)
    - `lang="`*`language-code-AVT`*`"` (default "`en`" for text data type in XSLT 1.0)
      - override Unicode code point order with a specific language
      - as specified by XML 1.0 recommendation for the `xml:lang` attribute
        - IETF RFC 1766 "Tags for the Identification of Languages"
    - `case-order="`*`AVT`*`"` and default are language dependent
      - `case-order="upper-first"`
        - indicates the sort order is `A a B b C c`
      - `case-order="lower-first"`
        - indicates the sort order is `a A b B c C`
    - W3C I18N Working Group
      - responsible for issues of internationalization
      - results of work will be incorporated into XSLT
  - `data-type="number"`
    - indicates the sort order is numeric based on the text of the sort keys being converted to numeric values (the `lang=` attribute is ignored)
  - 2 `data-type="`*`XPath-2-data-type`*`"` (default basis in XSLT 2.0)
    - e.g. `data-type="xs:dateTime"`
  - `data-type="`*`prefix`*`:`*`processor-recognized-sort-scheme-name`*`"`
    - `xmlns:`*`prefix`*`="`*`processor-recognized-URI-reference`*`"`
    - indicates the sort order is according to an algorithm recognized by the XSLT processor for the specified scheme

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sort examples

An example with multiple sort keys:

- people's names are composite (given name and surname), but treated as a whole
  - both parts of the name are always displayed when the name is displayed
- the first part of the report is in names in document order
- the second part of the report is the names sorted by the surname
- the third part of the report is the names sorted first by the surname and then where equal by the given name

The example XML source `sorttest.xml` is as follows:

```
01 <?xml version="1.0"?>
02 <names>
03 <name><given>Julie</given><surname>Holman</surname></name>
04 <name><given>Margaret</given><surname>Mahoney</surname></name>
05 <name><given>Ted</given><surname>Holman</surname></name>
06 <name><given>John</given><surname>Mahoney</surname></name>
07 <name><given>Kathryn</given><surname>Holman</surname></name>
08 <name><given>Ken</given><surname>Holman</surname></name>
09 </names>
```

The desired result is as follows:

```
01 Holman,Julie
02 Mahoney,Margaret
03 Holman,Ted
04 Mahoney,John
05 Holman,Kathryn
06 Holman,Ken
07 ---
08 Holman,Julie
09 Holman,Ted
10 Holman,Kathryn
11 Holman,Ken
12 Mahoney,Margaret
13 Mahoney,John
14 ---
15 Holman,Julie
16 Holman,Kathryn
17 Holman,Ken
18 Holman,Ted
19 Mahoney,John
20 Mahoney,Margaret
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Sort examples (cont.)

The following stylesheet `sorttest.xsl` will produce the desired results:

```
<?xml version="1.0"?><!--sorttest.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;">]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:apply-templates select="/*/name"/>         <!--unsorted-->
  <xsl:text>---&nl;</xsl:text>
  <xsl:apply-templates select="/*/name"> <!--sort with one key-->
    <xsl:sort select="surname"/>
  </xsl:apply-templates>
  <xsl:text>---&nl;</xsl:text>
  <xsl:for-each select="/*/name">       <!--sort with two keys-->
    <xsl:sort select="surname"/>
    <xsl:sort select="given"/>
    <xsl:call-template name="name"/>         <!--show the name-->
  </xsl:for-each>
</xsl:template>

          <!--both direct and indirect application of template-->
<xsl:template name="name" match="name">
  <xsl:value-of select="surname"/>,<xsl:value-of select="given"/>
  <xsl:text>&nl;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- the name display template is used for both push and pull operations on the tree
- the program illustrates pushing with one sort key and pulling with two sort keys

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping objectives

Reporting a collection of information grouped by components with equal values

- simple sorting obliges the reporting of all information in the component
- grouping requires identifying all items with a common component and reporting under each unique value of that component
  - identify unique values of grouping component
  - report other components of all items with same grouping value

Types of grouping:

- adjacent grouping
  - suppression of common adjacent values
- uniqueness grouping
  - identify unique values in a set of nodes
  - group the set of nodes according to unique values
    - show the common use of the unique value as a heading of the members of the set
  - list the sub-constructs under the unique values as distinguishers

XSLT 1.0 approaches for grouping and uniqueness are algorithmic

- requires coding of the approach by the transform writer

XSLT 2.0 approaches are declarative using built-in facilities

P2 General-purpose facility for uniqueness available in XPath 2.0

- finding distinct values: `distinct-values(`*`items`*`)`
- work with a set of unique values found in the collection of items
  - uniqueness determined by string value comparison

T2 General-purpose facility for grouping available in XSLT 2.0

- finding the first of groups of distinct nodes or values: `<xsl:for-each-group>`
- work with groups of items returned based upon grouping criteria

T1 It is often difficult to characterize a problem to be solved as being a grouping problem

- for example, consider "finding all the countries covered by sales territories by different salesmen"
- a data set might contain 100 salesmen records, each indicating the country in which they sell
- the report desired need only list those countries which are covered, not including a list of the salesmen for each country
- finding the unique set of countries is the first step of the grouping problem, the second step of grouping isn't engaged
- not difficult in XSLT 2.0 because of the available functions

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Adjacent grouping in XSLT 1.0

Axis-based tests for the first in document order or for closest preceding construct:

- checks for presence/absence of a prior occurring construct
- recall the axis directions in XPath shown on slide 72

To test preceding sibling values:

- `preceding-sibling::`*`test`*`[1]`
  - *always* uses reverse document order of nodes
    - axes are source node tree proximity oriented
    - the first item on prior axes is the closest such item
      - if there are no such first items, then the current node must be the first item
  - this pattern *does not* act on resulting sort order of nodes
    - there are no functions to work on or address members of a sorted set of nodes

To test preceding values when not dealing with siblings:

- `preceding::`*`test`*`[1]`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Adjacent grouping in XSLT 1.0 (cont.)

The example XML source `sorttest.xml` is as follows:

```
01 <?xml version="1.0"?>
02 <names>
03 <name><given>Julie</given><surname>Holman</surname></name>
04 <name><given>Margaret</given><surname>Mahoney</surname></name>
05 <name><given>Ted</given><surname>Holman</surname></name>
06 <name><given>John</given><surname>Mahoney</surname></name>
07 <name><given>Kathryn</given><surname>Holman</surname></name>
08 <name><given>Ken</given><surname>Holman</surname></name>
09 </names>
```

Consider the need to produce the following result, the original and sorted summary, grouped omitting adjacent surnames:

```
01 Holman, Julie
02 Mahoney, Margaret
03 Holman, Ted
04 Mahoney, John
05 Holman, Kathryn
06 Holman, Ken
07 ---
08 Holman    Julie
09 Mahoney   Margaret
10 Holman    Ted
11 Mahoney   John
12 Holman    Kathryn
13           Ken
```

Note above how only the last prefix is omitted because the only adjacent common surnames are at the end of the original list of names.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Adjacent grouping in XSLT 1.0 (cont.)

The following script `groupadj.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--groupadj.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="/*/name">                  <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text></xsl:for-each>
  <xsl:text>---&nl;</xsl:text>

  <xsl:for-each select="/*/name">           <!--group adjacent-->
    <xsl:choose>
      <xsl:when test="preceding-sibling::name[1]
                                [surname=current()/surname] )">
        <xsl:text>&pad;</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="surname"/>
        <xsl:value-of select="substring( '&pad;',
                              string-length( surname ) + 1 )"/>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:value-of select="given"/>
    <xsl:text>&nl;</xsl:text>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- when checking for like surnames when adjacent:
  - the test first checks if the node is the first node of the set by looking for any preceding sibling element nodes of the same name
  - if there are preceding sibling element nodes, then the presence of immediately preceding sibling element node with the same value of the surname element child is checked
- each displayed surname is followed by the number of spaces needed to fill out the padding
  - this creates a virtual tab stop in the text output

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

The essence of grouping under uniqueness

The essence of grouping under uniqueness is to utilize a test on all values that is true only once for each unique value

- recall that sorting does not rearrange the source node tree and that axes always relate to the order in the source node tree
- could check along an axis in either document order to check following values or reverse document order to find preceding values
- for example purposes, the following will check the set of nodes in reverse document order to be empty to determine a given value is the first in document order

1 For each sort key, find the unique data values for that key:

- visit all values in the population to be grouped for the key
  - the visiting may be done in sequence order or in sorted order
  - determine for each given value whether it is the first such value in sequence order in the source node tree
  - the source tree can be tested for any preceding nodes of the same calculated comparison value
- at the first in sequence order for each value, handle the group with that value
  - if there are no preceding nodes in the source node tree of the same value assume the given value has not been processed yet
  - process the given value
    - revisit data selecting only those nodes matching the given value
    - process the resulting current node set that represents the key's value set
      - for example, do next level sort given the value found at this level
      - for example, do the final result tree building based on key value
- after the first in sequence order for each value, do nothing
  - if there are preceding nodes in the source node tree of the same value assume the given value has already been processed
  - ignore the node and value

2 Built-in facilities for grouping

- establish the grouping keys found in the population based on grouping criteria
  - four different methods of stating criteria
- for each group defined by its grouping key act on the first in the group
  - access is available to the current grouping key
  - the current value is the first of the population with that value
- within each group process the members of the group as a set
  - access is available to the current group where all members have the same grouping key

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

The essence of grouping under uniqueness (cont.)

Consider the unsorted set of surnames, where the `generate-id()` value is exposed and first of each surname is identified:

```
[Unsorted surnames]
1: Holman, Julie (d0e3) - First "Holman"
2: Mahoney, Margaret (d0e9) - First "Mahoney"
3: Holman, Ted (d0e15)
4: Mahoney, John (d0e21)
5: Holman, Kathryn (d0e27)
6: Holman, Ken (d0e33)
```

All of the surnames are visited, but action is taken only on the first of each unique surname value; the given names are revisited in sorted order of given name:

```
[Visit all of the surnames]
1: Holman, Julie (d0e3) - First "Holman" d0e3=d0e3
          [Revisit all of the same surname]
          1: Holman, Julie (d0e3)
          2: Holman, Kathryn (d0e27)
          3: Holman, Ken (d0e33)
          4: Holman, Ted (d0e15)
2: Mahoney, Margaret (d0e9) - First "Mahoney" d0e9=d0e9
          [Revisit all of the same surname]
          1: Mahoney, John (d0e21)
          2: Mahoney, Margaret (d0e9)
[3: Holman, Ted (d0e15)] - Not first "Holman" d0e15!=d0e3
[4: Mahoney, John (d0e21)] - Not first "Mahoney" d0e21!=d0e9
[5: Holman, Kathryn (d0e27)] - Not first "Holman" d0e27!=d0e3
[6: Holman, Ken (d0e33)] - Not first "Holman" d0e33!=d0e3
```

Important note: the uniqueness and grouping algorithms work even if the selections are not sorted

- When using XSLT 2.0 only the first in the population of each distinct value is visited
  - in this example it would be "`Holman (d0e3)`" and "`Mahoney (d0e9)`"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using axes in XSLT 1.0

Consider the need to produce the following result, the original and sorted summary, grouped according to surname, using axes to determine the essence of grouping under uniqueness:

```
Holman, Julie
Mahoney, Margaret
Holman, Ted
Mahoney, John
Holman, Kathryn
Holman, Ken
---
Holman:
          Julie
          Kathryn
          Ken
          Ted
Mahoney:
          John
          Margaret
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using axes in XSLT 1.0 (cont.)

The following script `group.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--group.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="/*/name">                  <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text></xsl:for-each>
  <xsl:text>---&nl;</xsl:text>

  <xsl:for-each select="/*/name">              <!--group sorted-->
    <xsl:sort select="surname"/>              <!--by primary key-->

     <!--the following test is only true once for each primary
     key, even though all elements are visited by the for-each-->
    <xsl:if test="not( preceding-sibling::name
                                  [surname=current()/surname] )">
      <xsl:value-of select="surname"/>
      <xsl:text>:&nl;</xsl:text>
              <!--reselect only nodes with current primary key-->
      <xsl:for-each select="/*/name[surname=current()/surname]">
        <xsl:sort select="given"/>          <!--by secondary key-->
        <xsl:text>&pad;</xsl:text>
        <xsl:value-of select="given"/>
        <xsl:text>&nl;</xsl:text>
      </xsl:for-each>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- every node in the data is repeatedly visited once for each node of the data
  - the number of visits is the number of nodes squared
  - once to establish order of the primary key
  - once to establish order of the secondary key within all elements matching the primary key

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using variables in XSLT 1.0

Given that finding required nodes in a source node tree when using axes could be comparatively slow as the tree grows larger, using variables can extract the nodes into a collection that can be searched faster than when using the tree.

Grouping by identifying unique values within variables of node-sets:

- collect the population needing to be grouped into a node-set variable
  - the nodes can come from different node trees
- find the first member in the variable with each unique value
  - show the unique value of the construct as a heading
- reselect the members from the variable using that construct's value, displaying distinguishing values separately
  - selecting from a variable is faster than selecting from the tree

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using variables in XSLT 1.0 (cont.)

Note first that the generated id can easily be obtained of the first member in the variable of a given value

- `generate-id($`*`variable`*`[`*`value-predicate`*`][1])`
- recall that the generated identifier for a node is persistent through the execution of a given stylesheet and that it operates on the first member of a given node set
  - two nodes can be checked as being the same node by comparing their respective generated identifiers
  - the use of "`[1]`" above is redundant but helpful when learning

For each key, starting with the primary sort key:

- extract all items that are to be grouped into a variable
  - if desired, sort the items and examine each one in the variable in sorted order
  - otherwise, just examine each variable in adjacent order to get an adjacent grouping
- only act on the determination of "this node is the first in the variable with its value"
  - `generate-id(.) = generate-id($`*`variable`*`[`*`value-predicate`*`][1])`
  - skip processing on all other variable members that do not pass the test because they aren't the first of their respective values
- group-oriented processing can occur for each node selected as it represents each unique value from the variable
- reselect all nodes in the variable matching the particular value
  - `select="$`*`variable`*`[`*`value-predicate`*`]"`
- process the resulting current node set that represents the sort key's unique value set of nodes
  - for example, do next level sort given the value found at this level
  - for example, do the final result tree building based on key value

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using variables in XSLT 1.0 (cont.)

The following script `group2.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--group2.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">                              <!--root rule-->
  <xsl:for-each select="/*/name">                      <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text></xsl:for-each>
  <xsl:text>---&nl;</xsl:text>
                                                    <!--group sorted-->
  <xsl:variable name="names" select="/*/name"/>
  <xsl:for-each select="$names">          <!--look at all names-->
    <xsl:sort select="surname"/>             <!--by primary key-->

      <!--only act if this is the first node with this surname-->
    <xsl:if test="generate-id(.)=
                  generate-id($names[surname=current()/surname][1])">
      <xsl:value-of select="surname"/>
      <xsl:text>:&nl;</xsl:text>
                          <!--get all nodes for the given surname-->
      <xsl:for-each select="$names[surname=current()/surname]">
        <xsl:sort select="given"/>         <!--by secondary key-->
        <xsl:text>&pad;</xsl:text>
        <xsl:value-of select="given"/>
        <xsl:text>&nl;</xsl:text>
      </xsl:for-each>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using keys in XSLT 1.0

Given that access to a key table should be optimized by a processor, obtaining nodes based on a lookup value should be faster than either source tree access or variable access.

Grouping by identifying key nodes

- group a set of nodes according to a node's presence in an `<xsl:key>` table
- find the first member in the table with each unique value
  - show the unique value of the construct as a heading
- reselect the members from the table using that construct's value, displaying distinguishing values separately
  - selecting from the table is faster than selecting from the tree or from a variable

Colloquially referred to as the "Muenchian Method" after Steve Muench who first proposed the use of `<xsl:key>` for grouping.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using keys in XSLT 1.0 (cont.)

Note first that the generated id can easily be obtained of the first member in the table of a given lookup value

- `generate-id(key(`*`key-qname`*`,`*`lookup-value`*`)[1])`
- recall that the generated identifier for a node is persistent through the execution of a given stylesheet and that it operates on the first member of a given node set
  - two nodes can be checked as being the same node by comparing their respective generated identifiers
  - the above use of `"[1]"` is redundant

For each key, starting with the primary key:

- find unique values of the key:
  - find the generated id of the first entry of a given key's value in the `<xsl:key>` table
- find each node in the tree that is the first node of the key's unique value:
  - when used as a predicate, the following expression finds all first matching members of a location step resulting from the node test
  - `select="`*`axis-and-node-test`*`[generate-id(.)= generate-id(key(`*`key-qname`*`,`*`lookup-value`*`)[1])]"`
- the found nodes can be sorted if desired
- group-oriented processing can occur for each node selected as it represents each unique value from the key table
- reselect all nodes in the tree matching the particular value:
  - `select="key(`*`key-qname`*`,`*`lookup-value`*`)"`
- process the resulting current node set that represents the key's value set
  - for example, do next level sort given the value found at this level
  - for example, do the final result tree building based on key value

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using keys in XSLT 1.0 (cont.)

The example XML source `sorttest.xml` is as follows:

```
<?xml version="1.0"?>
<names>
<name><given>Julie</given><surname>Holman</surname></name>
<name><given>Margaret</given><surname>Mahoney</surname></name>
<name><given>Ted</given><surname>Holman</surname></name>
<name><given>John</given><surname>Mahoney</surname></name>
<name><given>Kathryn</given><surname>Holman</surname></name>
<name><given>Ken</given><surname>Holman</surname></name>
</names>
```

Consider yet again the need to produce the following result, the original plus a sorted summary grouped according to surname:

```
Holman, Julie
Mahoney, Margaret
Holman, Ted
Mahoney, John
Holman, Kathryn
Holman, Ken
---
Holman:
          Julie
          Kathryn
          Ken
          Ted
Mahoney:
          John
          Margaret
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness using keys in XSLT 1.0 (cont.)

The following script `group3.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--group3.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:key name="names" match="name" use="surname"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="/*/name">                  <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text></xsl:for-each>
  <xsl:text>---&nl;</xsl:text>
                                                 <!--group sorted-->
                        <!--get the first node of each surname-->
  <xsl:for-each select="/*/name
          [generate-id(.)=generate-id(key('names',surname)[1])]">
    <xsl:sort select="surname"/>            <!--by primary key-->
    <xsl:value-of select="surname"/>
    <xsl:text>:&nl;</xsl:text>
                       <!--get all nodes for the given surname-->
    <xsl:for-each select="key('names',surname)">
      <xsl:sort select="given"/>          <!--by secondary key-->
      <xsl:text>&pad;</xsl:text>
      <xsl:value-of select="given"/>
      <xsl:text>&nl;</xsl:text>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness within sub-trees in XSLT 1.0

T1 Typical use of the key table scopes key values across the entire document

- the key() function in XSLT 1.0 does not access only a sub-tree

Consider the following data in `grpsub.xml`:

- course contains lessons
- lessons contain frames
- frames distinguished by their applicability as stated in an attribute
- lessons and frames have titles authored by hand (note the white space)

```
<?xml version="1.0"?>
<course>
<lesson>
<title>
The XML family of Recommendations</title>
<frame id="xml-info" appl="a">
<title>
Extensible Markup Language (XML)</title>
</frame>
<frame id="xml-links" appl="b">
<title>
XML information links</title>
</frame>
...
</lesson>
<lesson>
<title>
Transformation data flows</title>
<frame id="x2xml" appl="a">
<title>
Transformation from XML to XML</title>
</frame>
...
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

The information grouped across the document, alphabetically by applicability, is as follows:

```
a
   Extensible Markup Language (XML) 1.1
   Some simple examples 3.1
   Stylesheet association 1.10
   Stylesheet requirements 4.1
   Styling structured information 1.4
   Templates and template rules 4.3
   Three-tiered architectures 2.5
   Transformation from XML to XML 2.1
b
   Extensible Stylesheet Language Transformations (XSLT) 1.6
   Historical development of the XSL and XSLT Recommendations 1.7
   Instructions and literal result elements 4.2
   Transformation from XML to XSL formatting semantics 2.2
   XML information links 1.2
   XML Path Language (XPath) 1.3
   XSL information links 1.8
   XSLT as an application front-end 2.4
c
   Approaches to stylesheet design 4.6
   Explicitly declared stylesheets 4.5
   Extensible Stylesheet Language (XSL) 1.5
   Implicitly declared stylesheets 4.4
   Namespaces 1.9
   Transformation from XML to non-XML 2.3
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

To group across the document, the key value is a simple value calculated from relative node content

- use applicability attribute value as grouping criterion

```
<?xml version="1.0"?><!--grpsub.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;">]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:key name="appls" match="frame" use="@appl"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select=
 "//frame[generate-id(.)=generate-id(key('appls',@appl)[1])]">
    <xsl:sort select="@appl"/>    <!--alphabetical group order-->
    <xsl:text>  </xsl:text>
    <xsl:value-of select="@appl"/>       <!--show group title-->
    <xsl:text>&nl;</xsl:text>
    <xsl:for-each select="key('appls',@appl)"><!--all in group-->
      <xsl:sort select="normalize-space(title)"/>
      <xsl:text>    </xsl:text>
      <xsl:value-of select="normalize-space(title)"/>
      <xsl:text> </xsl:text>
      <xsl:number count="lesson|frame" level="multiple"/>
      <xsl:text>&nl;</xsl:text>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

Chapter 9 - Sorting and grouping
Section 2 - Grouping constructs found in information

The information grouped under each lesson, alphabetically by lesson and then alphabetically by applicability, is as follows:

- the lesson information is not composite, so grouping isn't needed
- the applicability information is composite, so grouping is needed

```
Stylesheet examples
  a
    Some simple examples 3.1
Syntax basics - stylesheets, templates, instructions
  a
    Stylesheet requirements 4.1
    Templates and template rules 4.3
  b
    Instructions and literal result elements 4.2
  c
    Approaches to stylesheet design 4.6
    Explicitly declared stylesheets 4.5
    Implicitly declared stylesheets 4.4
The XML family of Recommendations
  a
    Extensible Markup Language (XML) 1.1
    Stylesheet association 1.10
    Styling structured information 1.4
  b
    Extensible Stylesheet Language Transformations (XSLT) 1.6
    Historical development of the XSL and XSLT Recommendations 1.7
    XML information links 1.2
    XML Path Language (XPath) 1.3
    XSL information links 1.8
  c
    Extensible Stylesheet Language (XSL) 1.5
    Namespaces 1.9
Transformation data flows
  a
    Three-tiered architectures 2.5
    Transformation from XML to XML 2.1
  b
    Transformation from XML to XSL formatting semantics 2.2
    XSLT as an application front-end 2.4
  c
    Transformation from XML to non-XML 2.3
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

To group under a sub-tree using variables, only those items in the sub-tree are subject to the algorithm:

```
<?xml version="1.0"?><!--grpsub2.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;"> ]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="//lesson"> <!--group under each lesson-->
    <xsl:sort select="normalize-space(title)"/>
    <xsl:value-of select="normalize-space(title)"/>
    <xsl:text>&nl;</xsl:text>

    <xsl:variable name="frames" select="frame"/>
    <xsl:for-each select="$frames">               <!--each group-->
      <xsl:sort select="@appl"/>  <!--alphabetical group order-->
      <xsl:variable name="appl" select="@appl"/>
      <xsl:if test="generate-id(.)=
                    generate-id($frames[@appl=$appl])">
        <xsl:text>  </xsl:text>
        <xsl:value-of select="@appl"/>    <!--show group title-->
        <xsl:text>&nl;</xsl:text>
        <xsl:for-each select="$frames[@appl=$appl]"><!--in grp-->
          <xsl:sort select="normalize-space(title)"/>
          <xsl:text>    </xsl:text>
          <xsl:value-of select="normalize-space(title)"/>
          <xsl:text> </xsl:text>
          <xsl:number count="lesson|frame" level="multiple"/>
          <xsl:text>&nl;</xsl:text>
        </xsl:for-each>
      </xsl:if>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

When working with key tables, values need not be solely constructed from node content

- values can be concatenated values from other sources relative to the node or generated from the node
- care must be taken to prevent ambiguous value calculations
  - the resulting concatenation of two values must not conflict with a third value
- `generate-id()` produces a lexical name token that cannot have spaces
  - a space is a viable delimiter of generated identifier values
- build table entry as a concatenation of three values:
  - generated identifier of the ancestral point in the hierarchy
  - the space
    - guaranteed to not be in the generated identifier
  - distinguishing node value
  - must lookup the table with the ancestral generated identifier or node value will not be found

Use of internal general entities can help work with complex code

- the algorithm for generating a key table node entry's value can be declared once and reused
- important for readability
- important for maintainability

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Grouping under uniqueness within sub-trees in XSLT 1.0 (cont.)

To group under a sub-tree using XSLT keys, the key value is a calculated from the identifier of the apex node combined with the relative node content (note the use of internal parsed general entities to promote consistency):

```
<?xml version="1.0"?><!--grpsub3.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [<!ENTITY nl "&#xd;&#xa;">
   <!--group under ancestral lesson by applicability attribute-->
<!ENTITY frame-lookup
     "concat( generate-id(ancestor::lesson),' ',@appl )">
<!ENTITY frame-group "key('appls', &frame-lookup;)">
<!ENTITY frame-group-first
        "frame[generate-id(.)=generate-id(&frame-group;[1])]"> ]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:key name="appls" match="frame" use="&frame-lookup;"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="//lesson"> <!--group under each lesson-->
    <xsl:sort select="normalize-space(title)"/>
    <xsl:value-of select="normalize-space(title)"/>
    <xsl:text>&nl;</xsl:text>
    <xsl:for-each select="&frame-group-first;"> <!--each group-->
      <xsl:sort select="@appl"/>  <!--alphabetical group order-->
      <xsl:text>  </xsl:text>
      <xsl:value-of select="@appl"/>      <!--show group title-->
      <xsl:text>&nl;</xsl:text>
      <xsl:for-each select="&frame-group;">   <!--all in group-->
        <xsl:sort select="normalize-space(title)"/>
        <xsl:text>    </xsl:text>
        <xsl:value-of select="normalize-space(title)"/>
        <xsl:text> </xsl:text>
        <xsl:number count="lesson|frame" level="multiple"/>
        <xsl:text>&nl;</xsl:text>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

When to use different grouping methods

- When using XSLT 1.0
  - adjacent grouping
    - use the axis-based method
    - only need to look at adjacent members of the set for a group
  - uniqueness grouping with single document-wide context
    - use either the variable-based or the key-based method
    - if the implementation is optimized, the key-based method *might* be a bit faster than the variable-based method
    - the variable-based method might be more intuitive to the reader
  - uniqueness grouping with partial-document context
    - use the variable-based method
    - easy to set the context of the grouping to any subset of the tree by addressing the desired nodes in a variable binding
    - establishing the uniqueness criteria for key-based grouping may be too awkward or time-consuming and may make the resulting logic too difficult to maintain
  - uniqueness grouping with multiple-document context
    - use the variable-based method
    - easy to set the context of the grouping to any set of nodes collected from multiple trees by using `document()`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Built-in grouping facilities in XSLT 2.0

```
<xsl:for-each-group select="population-expression"
                    ...grouping-criteria...>
  ...multiple <xsl:sort> instructions...
  ...template...
</xsl:for-each-group>
```

The objective is to find the first member of each group based on grouping criteria

- `select=` attribute specifies which nodes or values make up the population
- exactly one of the following four grouping schemes must be chosen
  - `group-adjacent="`*`expression`*`"`
    - creates a single grouping key based on adjacent values being equal strings
    - an error is signaled if the expression does not return a sequence of exactly one item per member
  - `group-by="`*`sequence-expression`*`"`
    - calculate one grouping value for each expression in the sequence
      - the item is placed in as many groups as there are expressions
    - when the resulting groups are visited, a given item might be found in up to as many groups as there are members of the sequence
  - `group-starting-with="`*`match-pattern`*`"`
    - nodes in the population are examined in population order
    - each successful match to given pattern starts a new group for the current node
    - the node is then added to the current group
  - `group-ending-with="`*`match-pattern`*`"`
    - nodes in the population are examined in population order
    - the node is then added to the current group
    - each successful match to given pattern starts a new group for the next node
- `collation="`*`uri-string-AVT`*`"` overrides any default collation
  - used when comparing strings to be equal or not

`group-starting-with=` and `group-ending-with=` are suited to inferring structure from flat information

- where groups of items are separated by sibling elements rather than by parent elements
- e.g. XHTML heading elements are siblings of content under the heading, not parents of content under the heading

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Built-in grouping facilities in XSLT 2.0 (cont.)

Any `<xsl:sort>` in `<xsl:for-each-group>` sorts the processing of the groups

- does not affect the processing of the members of the group

The context item inside the `<xsl:for-each-group>` is the first member of the group in population order

- one can address values off of the first member

Functions available when grouping:

- these functions have no return values outside of the context of a `<xsl:for-each-group>`

`current-grouping-key()`

- returns the value that forms the basis of the group
- prevents having to recalculate the `group-by=` expression

`current-group()`

- returns a guaranteed non-empty sequence of items found in the group associated with the `current-grouping-key()` value
- typically accessed using `<xsl:for-each>` possibly containing `<xsl:sort>`instructions for the group members
- items in the group are returned in input sequence order
- beware rearranging the items inadvertently:
  - ```
    <xsl:for-each select="current-group()">
      <xsl:value-of select="@p"/>
    </xsl:for-each>
    ```

    vs

    ```
    <xsl:value-of select="current-group()/@p"/>
    ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.
    ```

Adjacent grouping in XSLT 2.0

Chapter 9 - Sorting and grouping
Section 2 - Grouping constructs found in information

The example XML source `sorttest.xml` is as follows:

```
<?xml version="1.0"?>
<names>
<name><given>Julie</given><surname>Holman</surname></name>
<name><given>Margaret</given><surname>Mahoney</surname></name>
<name><given>Ted</given><surname>Holman</surname></name>
<name><given>John</given><surname>Mahoney</surname></name>
<name><given>Kathryn</given><surname>Holman</surname></name>
<name><given>Ken</given><surname>Holman</surname></name>
</names>
```

Consider the need to produce the following result, the original and sorted summary, grouped omitting adjacent surnames:

```
Holman, Julie
Mahoney, Margaret
Holman, Ted
Mahoney, John
Holman, Kathryn
Holman, Ken
---
Holman    Julie
Mahoney   Margaret
Holman    Ted
Mahoney   John
Holman    Kathryn
          Ken
```

Note above how only the last prefix is omitted because the only adjacent common surnames are at the end of the original list of names.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Adjacent grouping in XSLT 2.0 (cont.)

Chapter 9 - Sorting and grouping
Section 2 - Grouping constructs found in information

The following script `groupx2adj.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--groupx2adj.xsl-->
<!--XSLT 2.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each select="/*/name">                  <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text>
  </xsl:for-each>
  <xsl:text>---&nl;</xsl:text>

  <xsl:for-each-group select="/*/name"
                      group-adjacent="surname"> <!--by surname-->
   <!--first in the group must be different from previous-->
   <xsl:value-of select="concat(surname,
            substring('&pad;',string-length(surname)+1),given)"/>
   <xsl:text>&nl;</xsl:text>
   <xsl:for-each select="current-group()[position()>1]">
     <!--omit surname if not the first in adjacent group-->
     <xsl:value-of select="concat('&pad;',given)"/>
     <xsl:text>&nl;</xsl:text>
   </xsl:for-each>
  </xsl:for-each-group>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- each displayed surname is followed by the number of spaces needed to fill out the padding string
  - this creates a virtual tab stop in the text output

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Adjacent grouping in XSLT 2.0 (cont.)

Unobvious criteria can be used for adjacent grouping

- any expression can be used in `group-adjacent=`
- the resulting value is associated with the node being grouped

Consider the need to create pairs of nodes from any sequence of nodes

- use the same numeric value for adjacent nodes in the sequence
- group adjacent based on the numeric values

```
<table>
  <xsl:for-each-group select="items/item"
                      group-adjacent="(position()-1) idiv $cols">
    <row>
      <xsl:for-each select="current-group()">
        <col><xsl:value-of select="."/></col>
      </xsl:for-each>
    </row>
  </xsl:for-each-group>
</table>
```

From a sequence of items:

```
<items>
  <item>a</item>
  <item>b</item>
  <item>c</item>
  <item>d</item>
  <item>e</item>
</items>
```

To a sequence of paired items:

```
<table>
   <row>
      <col>a</col>
      <col>b</col>
   </row>
   <row>
      <col>c</col>
      <col>d</col>
   </row>
   <row>
      <col>e</col>
   </row>
</table>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness in XSLT 2.0

The following script `groupx2by.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--groupx2by.xsl-->
<!--XSLT 2.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:output method="text"/>

<xsl:template match="/">                              <!--root rule-->
  <xsl:for-each select="/*/name">                      <!--unsorted-->
    <xsl:value-of select="concat(surname,', ',given)"/>
    <xsl:text>&nl;</xsl:text></xsl:for-each>
  <xsl:text>---&nl;</xsl:text>

  <xsl:for-each-group select="/*/name"
                      group-by="surname">     <!--group sorted-->
    <xsl:sort select="surname"/>            <!--by primary key-->

    <xsl:value-of select="current-grouping-key()"/>
    <xsl:text>:&nl;</xsl:text>
              <!--reselect only nodes with current primary key-->
    <xsl:for-each select="current-group()">
      <xsl:sort select="given"/>        <!--by secondary key-->
      <xsl:text>&pad;</xsl:text>
      <xsl:value-of select="given"/>
      <xsl:text>&nl;</xsl:text>
    </xsl:for-each>
  </xsl:for-each-group>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- `select="current-grouping-key()"` could be `select="surname"`
  - identical result because of simple calculation of the grouping key

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping under uniqueness in XSLT 2.0 (cont.)

Each item in the population can be put into more than one group

- the `group-by=` attribute can be a sequence of multiple group value calculations
- the item shows up in each group based on the result of each calculation

```
<xsl:variable name="grouptest" as="element()+">
  <x id="a" val="1"/>
  <x id="b" val="2"/>
  <x id="c" val="3"/>
  <x id="d" val="4"/>
  <x id="e" val="5"/>
  <x id="f" val="6"/>
  <x id="g" val="7"/>
  <x id="h" val="8"/>
  <x id="i" val="9"/>
</xsl:variable>
<xsl:for-each-group select="$grouptest"
                    group-by="@val + 1, @val + 2, @val mod 4">
  <xsl:sort select="current-grouping-key()"/>
<xsl:text>
</xsl:text>
  <xsl:value-of select="current-grouping-key()"/>: <xsl:text/>
  <xsl:value-of select="current-group()/@id" separator=", "/>
</xsl:for-each-group>
```

Exposing the groups created shows the items in different groups

- e.g. the node with identifier "a" is in groups 1, 2 and 3

```
0: d, h
1: a, e, i
2: a, b, f
3: a, b, c, g
4: b, c
5: c, d
6: d, e
7: e, f
8: f, g
9: g, h
10: h, i
11: i
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping flat information in XSLT 2.0

The following script `groupx2flat.xsl` accomplishes the desired result:

```
<?xml version="1.0"?><!--groupx2flat.xsl-->
<!--XSLT 2.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
<!ENTITY pad "          ">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:output method="text"/>

<xsl:template match="/">                          <!--root rule-->
  <xsl:for-each-group select="/*/*"
                      group-starting-with="generation">
    <xsl:value-of select="."/>
    <xsl:text/>(<xsl:value-of select="count(current-group())-1"/>
    <xsl:text>):&nl;</xsl:text>
                  <!--reselect only those nodes that are names-->
    <xsl:for-each select="current-group()[self::name]">
      <xsl:sort select="surname"/>          <!--by primary key-->
      <xsl:sort select="given"/>          <!--by secondary key-->
      <xsl:value-of select="'  ',given,surname"/>
      <xsl:text>&nl;</xsl:text>
    </xsl:for-each>
  </xsl:for-each-group>
</xsl:template>

</xsl:stylesheet>
```

Of note:

- each group contains the node that triggered the group and the nodes that did not trigger the next group
  - requires the `count(current-group())-1` to reflect the count without the grouping trigger
  - `count(current-group[self::name])` would work just as well

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping flat information in XSLT 2.0 (cont.)

Consider the need to create a hierarchy from flat XHTML:

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Test</title></head>
  <body>
    <p>Outside of any level</p>
    <h1>First level 1</h1>
    <p>Level 1 text</p>
    <p>Level 1 text</p>
    <p>Level 1 text</p>
    <h2>Second level 1.1</h2>
    <p>Level 1.1 text</p>
    <p>Level 1.1 text</p>
    <p>Level 1.1 text</p>
    <h2>Second level 1.2</h2>
    <p>Level 1.2 text</p>
    <p>Level 1.2 text</p>
    <p>Level 1.2 text</p>
    <h1>First level 2</h1>
    <p>Level 2 text</p>
    <p>Level 2 text</p>
    <p>Level 2 text</p>
    <h2>Second level 2.1</h2>
    <p>Level 2.1 text</p>
    <p>Level 2.1 text</p>
    <p>Level 2.1 text</p>
    <h2>Second level 2.2</h2>
    <p>Level 2.2 text</p>
    <p>Level 2.2 text</p>
    <p>Level 2.2 text</p>
  </body>
</html>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Grouping flat information in XSLT 2.0 (cont.)

Chapter 9 - Sorting and grouping
Section 2 - Grouping constructs found in information

The objective is to create a hierarchy of the levels implied by `h1` and `h2`:

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <p>Outside of any level</p>
   <group>
      <title>First level 1</title>
      <p>Level 1 text</p>
      <p>Level 1 text</p>
      <p>Level 1 text</p>
      <group>
         <title>Second level 1.1</title>
         <p>Level 1.1 text</p>
         <p>Level 1.1 text</p>
         <p>Level 1.1 text</p>
      </group>
      <group>
         <title>Second level 1.2</title>
         <p>Level 1.2 text</p>
         <p>Level 1.2 text</p>
         <p>Level 1.2 text</p>
      </group>
   </group>
   <group>
      <title>First level 2</title>
      <p>Level 2 text</p>
      <p>Level 2 text</p>
      <p>Level 2 text</p>
      <group>
         <title>Second level 2.1</title>
         <p>Level 2.1 text</p>
         <p>Level 2.1 text</p>
         <p>Level 2.1 text</p>
      </group>
      <group>
         <title>Second level 2.2</title>
         <p>Level 2.2 text</p>
         <p>Level 2.2 text</p>
         <p>Level 2.2 text</p>
      </group>
   </group>
</result>
```

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Grouping flat information in XSLT 2.0 (cont.)

The supplied `groupx2flat2.xsl` accomplishes this task

```
<?xml version="1.0"?><!--groupx2flat2.xsl-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           xpath-default-namespace="http://www.w3.org/1999/xhtml"
           version="2.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">                          <!--root rule-->
  <result>
    <xsl:for-each-group select="/*/body/*"
                        group-starting-with="h1">
      <xsl:choose>   <!--check first member to see if in an h1-->
        <xsl:when test="not(self::h1)">       <!--not grouped-->
          <xsl:apply-templates select="current-group()"/>
        </xsl:when>
        <xsl:otherwise>                 <!--found an h1 group-->
          <group><title><xsl:apply-templates/></title>
            <xsl:for-each-group group-starting-with="h2"
                         select="current-group()[position()>1]">
              <xsl:choose>       <!--check to see if in an h2-->
                <xsl:when test="not(self::h2)"><!--not grouped-->
                  <xsl:apply-templates select="current-group()"/>
                </xsl:when>
                <xsl:otherwise>         <!--found an h2 group-->
                  <group><title><xsl:apply-templates/></title>
                    <xsl:apply-templates
                         select="current-group()[position()>1]"/>
                  </group>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:for-each-group>
          </group>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each-group>
  </result>
</xsl:template>

<xsl:template match="*">    <!--strip namespaces from elements-->
  <xsl:element name="{local-name(.)}">
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Grouping based on a concatenated sequence

XSLT grouping distinctions are based on singleton values, not sequences

- a sequence of values must be atomized to a single value
- how does one concatenate the sequence values accommodating absent members?

A sequence delimiter must separate each of the sequence members

- this prevents ambiguity of concatenated fields
- i.e. `('a','x')` needs to sort before `('ab','a')`

The Unicode `&#xfffd;` code point is a special "replacement character"

- used to signal an unknown character has been replaced with the replacement character
- should not be found in interchanged documents, so fairly safe to use
- sorts higher than every other Unicode character in the base multilingual plane

The carriage return `
` code point is rare in XML documents

- most carriage returns are parts of end-of-line sequences that are normalized to `
`
- should not be found in interchanged documents, so fairly safe to use
- sorts lower than every other printable Unicode character

Any concatenation needs to accommodate absent members as either "high values" or "low values" in the resulting string

- if an absent member is to sort after non-absent members:
  - use a high value in the delimiter, e.g. `&#xfffd;`
  - `group-by="concat(Heading,'&#xfffd;',`
`SubDiv1,'&#xfffd;',`
`SubDiv2,'&#xfffd;',SubDiv3)"`
- if an absent member is to sort before non-absent members:
  - use a low value in the delimiter, e.g. `
&#xfffd;`
  - `group-by="concat(Heading,'
&#xfffd;',`<br>`SubDiv1,'
&#xfffd;',`<br>`SubDiv2,'
&#xfffd;',SubDiv3)"`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Finding the minimum and maximum values

Chapter 9 - Sorting and grouping
Section 3 - Other uses of sorting in XSLT 1.0

Sometimes it is necessary to identify nodes that have a minimum or maximum value from a set of values

- recall a node-set comparison continues as long as the evaluation is false and stops once the evaluation is true
  - when used in a predicate as in the example below, only evaluates true for those nodes that are either largest or smallest based on the comparison
- recall that `<xsl:sort>` reorders the current node list (not the source node tree) and that `position()` reflects the sorted order (not the tree order)
  - the first item in the list is the first in sorted order

```
<?xml version="1.0"?><!--minmax.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY nl "&#xd;&#xa;">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">
  <xsl:variable name="itemnodes" select="//item"/>
  <xsl:text>Minimum nodes:&nl;</xsl:text>       <!--get nodes-->
  <xsl:for-each
        select="$itemnodes[ not( $itemnodes/@val &lt; @val ) ]">
    <xsl:value-of select="."/><xsl:text>&nl;</xsl:text>
  </xsl:for-each>

  <xsl:text>Maximum value: </xsl:text>   <!--get single value-->
  <xsl:for-each select="$itemnodes">    <!--sort all in order-->
    <xsl:sort data-type="number" order="descending"
              select="@val"/>
    <xsl:if test="position() = 1">   <!--first in list is max-->
      <xsl:value-of select="@val"/>
    </xsl:if>
  </xsl:for-each>
  <xsl:text>&nl;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Annex A - XML to HTML transformation

- Introduction - Historical web standards for presentation
- Section 1 - The W3C web presentation standards context
- Section 2 - Well-formed HTML
- Section 3 - HTML markup generation techniques

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Historical web standards for presentation

Annex A - XML to HTML transformation

Recognizing that the purpose of many XSLT transformations will be to render information over the World Wide Web, it is important to understand what different user-agent technologies are currently available to be used:

- user agents do not inherently understand the presentation semantics associated with our custom XML vocabularies
- can translate instances of our vocabularies into instances of a user agent vocabulary (e.g. HTML)
- can annotate instances of our vocabularies with formatting properties recognized by a user agent (e.g. CSS)

Hypertext Markup Language (HTML)

- a language for sharing text and graphics
- a hyperlinking facility for relating information

Cascading Stylesheets (CSS)

- getting away from the built-in user agent rendering semantics
- describes document tree ornamentation with formatting properties

User Agent Screen Painting

- direct control of the user agent canvas

Extensible Hypertext Markup Language (XHTML)

- modularization of HTML
- reformulating HTML as XML
- support of arbitrary XML in HTML

This annex overviews considerations for producing different flavors of HTML to support different user agents. As well, stylesheet fragments illustrating common requirements to mark up images and links are described.

Issues of compatibility between different user agent implementations and recommended markup practices are not reviewed in this material. A discussion of such issues can be found at `http://www.w3.org/TR/xhtml1/#guidelines`.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Hypertext Markup Language (HTML)

Annex A - XML to HTML transformation
Section 1 - The W3C web presentation standards context

Initially developed as a markup language for sharing information at the European Center for Nuclear Research (CERN) using a protocol called the Hypertext Transfer Protocol (HTTP) based on a vision of Tim Berners-Lee:

- project proposal within CERN - March 1989
  - `http://www.w3.org/History/1989/proposal.html`
- first published version of HTML representing initial practice in 1992
  - `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/MarkUp.html`
- first W3C version (HTML 2.0) came from HTML Working Group representing current practice in 1994
  - `http://www.w3.org/MarkUp/html-spec/` (edited by Dan Connolly)
- widely adopted version (HTML 3.2) representing current practice in 1996
  - `http://www.w3.org/TR/REC-html32.html` (edited by Dave Raggett)
- current W3C version (HTML 4.01) became a final recommendation in April 1998, revised in August 1999
  - `http://www.w3.org/TR/REC-html40/`
  - `http://www.w3.org/TR/1999/PR-html40-19990824`
  - introduces semantic-free constructs for browser canvas painting
  - introduces a document model manipulated by scripting for Dynamic HTML (DHTML)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Web Accessibility Initiative (WAI)

- `http://www.w3.org/WAI/`
- important guidelines for designing accessible web documents
- initially focused on HTML and user agents
- now five primary areas of focus
  - technology
  - guidelines
  - tools
  - education and outreach
  - research and development
- six sister groups
  - `http://www.w3.org/WAI/PF/` - protocol and formats
  - `http://www.w3.org/WAI/ER/` - evaluation and repair
  - `http://www.w3.org/WAI/GL/` - web content guidelines
  - `http://www.w3.org/WAI/UA/` - user agent guidelines
  - `http://www.w3.org/WAI/AU/` - authoring tool guidelines
  - `http://www.w3.org/WAI/EO/` - education and outreach
- specific XML-related guidelines
  - `http://www.w3.org/TR/xmlgl`

Web Content Accessibility Guidelines (WCAG)

- `http://www.w3.org/TR/WAI-WEBCONTENT`
- mandated in some European contexts

US Government Section 508 Compliance

- `http://www.section508.gov/`
- an amendment by Congress in 1998 to the Rehabilitation Act
- Federal agencies must make electronic and information technology accessible to people with disabilities

Bobby

- `http://www.cast.org/bobby`
- a free service to test web pages and expose barriers to accessibility

Tidy

- `http://www.w3.org/People/Raggett/tidy/`
- a free tool to report on and clean up HTML
- option "`-asxml`" to output XHTML

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Cascading Stylesheets (CSS)

HTML was not initially designed to be used as a general purpose browser canvas painting language

- it is a hypertext markup language describing links to information
- very difficult to flexibly paint the browser screen using the element types available
  - the built-in rendering semantics implemented in the browsers dictate the appearance

Vendors implemented incompatible extensions without sticking to recommendations

- the recommendations didn't provide the desired formatting at the time
- users wanted control of the canvas

Cascading style sheets were introduced

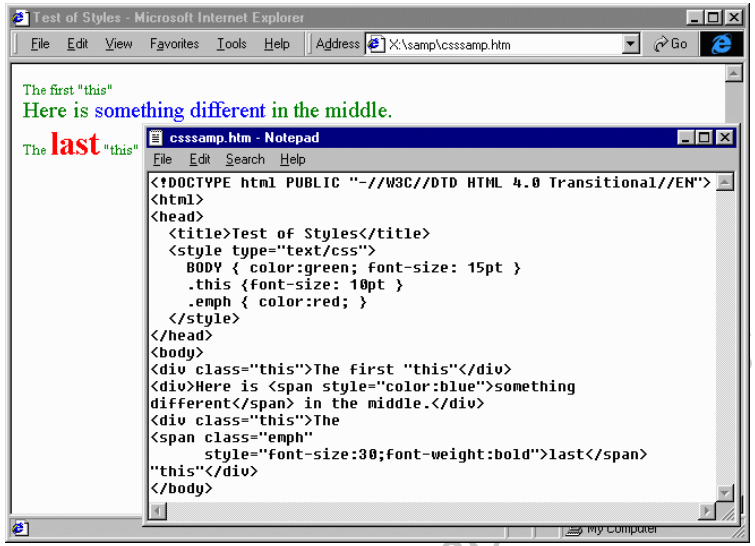
- targeted for both page designers and users (page readers)
- override built-in rendering semantics of browsers and achieve more of an artistic effect on the screen
- influence the presentation of documents without sacrificing device independence or adding new HTML element types
- decorate the document tree without modifying the structure of the document's hierarchy
- CSS-1 - December 1996
  - `http://www.w3.org/pub/WWW/TR/REC-CSS1`
  - adding style to web documents
  - a rule describes a facet of style; a collection of rules is a stylesheet
  - cascading allows the mixing of and overriding of stylesheets
- CSS-2 - May 1998
  - `http://www.w3.org/TR/REC-CSS2/`
  - media-specific stylesheets (e.g. printers and aural devices)
  - downloadable fonts
  - element positioning
  - tables
  - support for XML documents

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Browser screen painting

HTML 4.0 introduces the concept of Grouping Elements with no stand-alone presentational idioms (thus are semantic free):

- `<DIV>`
  - a block-level construct (breaks the line flow)
- `<SPAN>`
  - an inline-level construct (doesn't break the line flow)
- when used in conjunction with `id=` and `class=` attributes, adds generic method for adding structure to HTML presentation
- when used in conjunction with CSS style rule specifications, adds generic method for painting the browser screen with formatting properties
- the "cascade" is the priority of applying properties from different sources (e.g.: markup, identifier value, class value, element type property set, external stylesheet)



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Extensible HyperText Markup Language (XHTML)

- `http://www.w3.org/TR/xhtml1`
  - `http://www.w3.org/TR/xhtml-roadmap`
    - future plans for XHTML
  - `http://www.w3.org/TR/xhtml-events`
    - uniformly integrate behaviors of user agents
- a reformulation of HTML using XML 1.0
- reproduces, subsets and extends HTML 4 vocabulary
- XML-conforming and operates in HTML 4 conforming user agents
- XHTML files are acceptable input to XML processing tools
  - stylesheets
  - applications

IBTWSH

- `http://home.ccil.org/~cowan/XML/ibtwsh6.dtd`
- Itsy Bitsy Teeny Weeny Simple Hypertext DTD
  - written by John Cowan
- minimized document model of XHTML constructs useful for inclusion in other document models

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

What makes well-formed and valid HTML?

Empty elements in HTML, XML, and XHTML:

- e.g.: `<hr>`
  - according to HTML rules an empty element cannot be distinguished from a non-empty element by just the start tag
- e.g.: `<hr/>`
  - according to XML rules an empty element is distinguished by just the start tag
  - some older user agents regard the "`/`" as part of the element type and do not recognize or act on XML well-formed empty HTML elements
- e.g.: `<hr />`
  - according to XHTML guidelines for HTML user agents an empty element's name is followed by a space
    - this is not a normative requirement
  - satisfies those older user agents that regard the "`/`" as part of the element type
  - is well-formed XML but is not typically what is serialized by processors for the XML method

When the output method is HTML, the XSLT processor must serialize using HTML conventions:

- `<xsl:output method="html"/>`
- note also that the HTML output method is assumed when the output method is not specified and the document element of the result tree is "html" (in upper or lower case)
- known empty element types, attribute minimizations, built-in Latin1 character entity references, etc.

HTML completeness is important with respect to rigor

- older user agents cannot all accommodate a well-formed or valid HTML file
- `<!DOCTYPE`
  - should have declaration (though some older user agents incorrectly display this on the canvas)
  - note it is the responsibility of the stylesheet writer to use the `doctype-public=` attribute of `<xsl:output>` to ensure this
  - some user agents will accept incomplete HTML without a `<!DOCTYPE` declaration but will not accept incomplete HTML when there is a `<!DOCTYPE` declaration
- `<head>` and `<title>`
  - some older user agents require the content of the `<head><style>` element to be an HTML comment `<!-- -->` to prevent display on the canvas

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

What makes well-formed and valid HTML? (cont.)

The XHTML prologue and document element must be appropriately set

- some browsers will not recognize XHTML without all the necessary declarations
- some browsers still fail regardless of the correct declarations

Three possible subsets of XHTML available:

```
- <!DOCTYPE html
       PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
- <!DOCTYPE html
       PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
- <!DOCTYPE html
       PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

What makes well-formed and valid HTML? (cont.)

Must declare the default namespace in the document element:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

- some user agents won't accept other namespace declarations in the document element

Should use HTML media type when declaring the character encoding:

```
<?xml version="1.0" encoding="utf-8"?>
```

- XML declaration is to be exclusively used by XML compliant applications

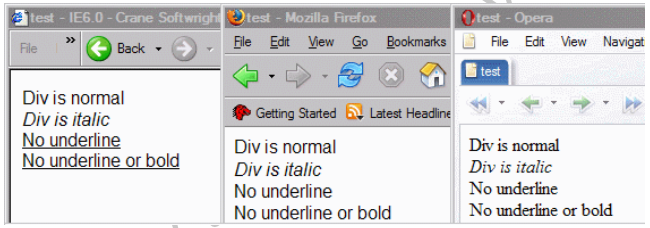
```
<meta http-equiv="Content-type" content='text/html; charset="utf-8"' />
```

- used by user agents that ignore the XML declaration may recognize this
- note the content type is not "`text/xml`" because it is assumed that an XML application would have already interpreted character the character encoding and the application must, therefore, be looking for an HTML encoding.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

What makes well-formed and valid HTML? (cont.)

Not all browsers correctly interpret empty tags




```
01 <?xml version="1.0" encoding="utf-8"?>
02 <!DOCTYPE html
03  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
04         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
05 <html xmlns="http://www.w3.org/1999/xhtml">
06 <head>
07  <title>test</title>
08  <meta http-equiv=
09        "Content-type" content='text/html; charset="utf-8"' />
10 </head>
11 <body>
12 <div>Div is normal</div>
13 <div style="font-style:italic">Div is italic</div>
14 <div style="text-decoration:underline" />
15 <div>No underline</div>
16 <div style="font-weight:bold"></div>
17 <div>No underline or bold</div>
18 </body>
19 </html>
```

Of note:

- an empty tag is interpreted in IE as a start tag and the formatting properties incorrectly continue through to the end of the document
  - shown in the example below with the setting of underline
- no problems when start and end tags are used instead of empty tags
  - shown in the example below with the setting of bold

1 2 Not a problem with the XHTML serialization

- the specification states that empty elements that are not declared empty cannot be minimized, thus forcing both the start and end tags

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Image elements

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

`<img>`

- empty element with two required attributes
- `src=`
  - points to the image storage location
- `alt=`
  - a short text description (important for accessibility; some user agents will display a pop-up of the text when the pointer hovers over the image)

Optional attributes:

- `longdesc=`
  - complements `alt=` by pointing to the Universal Resource Identifier (URI) of more information
- `height=` and `width=`
  - overriding the image parameters
- `usemap=` and `ismap=`
  - define client-side and server-side image maps (deprecated)

Note that while the HTML convention for displaying images is to point to the filename directly in the attribute value, and this can be done in XML, it is a common design practice to point to image files indirectly through unparsed entity declarations. An example of doing so is included in Chapter 7 Data type expressions and functions (page 270).

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

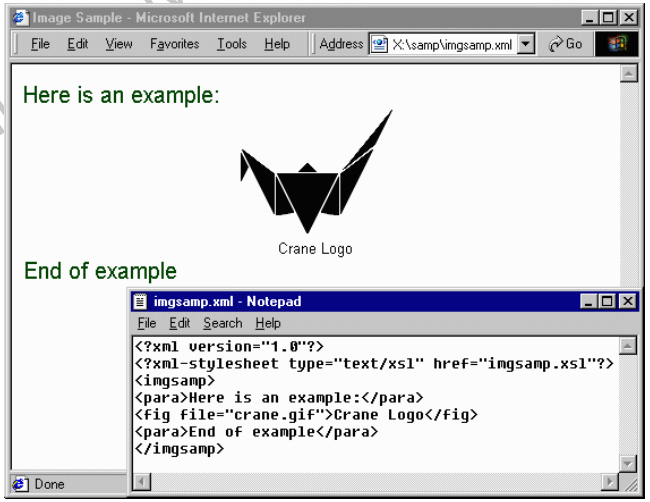
Image elements (cont.)

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

Consider the XML instance `imgsamp.xml` encoding the logo's filename directly in an attribute value:

```
01 <?xml version="1.0"?>
02 <?xml-stylesheet type="text/xsl" href="imgsamp.xsl"?>
03 <imgsamp>
04 <para>Here is an example:</para>
05 <fig file="crane.gif">Crane Logo</fig>
06 <para>End of example</para>
07 </imgsamp>
```

A possible rendering reveals the caption below the image itself:



Of note:

- the `fig` element type is not empty, the content is the figure caption
- the caption is being used both for the image alternate text and for the canvas

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Image elements (cont.)

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

One possible XSLT stylesheet, `imgsamp.xsl`, would be as follows:

```
<?xml version="1.0"?><!--imgsamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [ <!ENTITY dark-green "#004400"> ]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           version="1.0"
           xmlns="http://www.w3.org/TR/REC-html40">

<xsl:output method="html"
         doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"/>

<xsl:template match="/">                    <!-- root rule -->
  <html>
    <head><title>Image Sample</title></head>
    <body><xsl:apply-templates/></body>
  </html>
</xsl:template>

<xsl:template match="fig">
  <div style="font-size:10pt;text-align:center">
    <img src="{@file}" alt="{.}"/>
    <div>
      <xsl:apply-templates/>
    </div>
  </div>
</xsl:template>

<xsl:template match="para">
  <div style="font-size:15pt;color=&dark-green;">
    <xsl:apply-templates/>
  </div>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

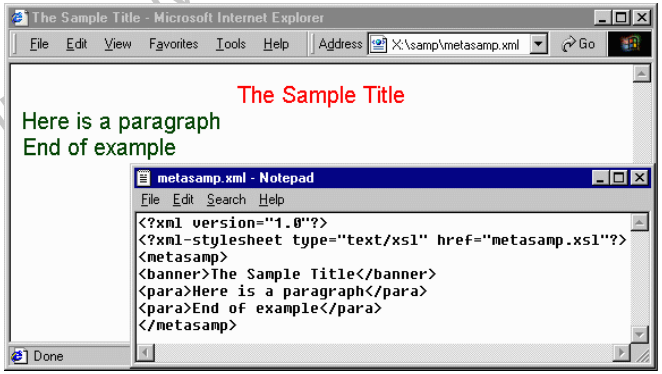
HTML meta-data

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

Consider the XML instance `metasamp.xml` encoding page title information in the instance:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="metasamp.xsl"?>
<metasamp>
<banner>The Sample Title</banner>
<para>Here is a paragraph</para>
<para>End of example</para>
</metasamp>
```

The following rendering reuses the meta data for the HTML file in two places, in each of the title bar of the window and on the canvas:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

HTML meta-data (cont.)

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

The XSL stylesheet `metasamp.xsl` renders the example:

```
<?xml version="1.0"?><!--metasamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [
<!ENTITY dark-green "#004400">
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">                      <!-- root rule -->
  <html>
    <head>
      <title><xsl:value-of select="//banner"/></title>
    </head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>

<xsl:template match="banner">
  <div style="font-size:15pt;color=red;text-align:center">
    <xsl:apply-templates/>
  </div>
</xsl:template>

<xsl:template match="para">
  <div style="font-size:15pt;color=&dark-green;">
    <xsl:apply-templates/>
  </div>
</xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Anchor elements

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

Consider the need to de-reference use of the XML concepts of ID and IDREF pointers (XML 1.0 recommendation production [56]) as in `asamp.xml`:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="asamp.xsl"?>
<!DOCTYPE asamp [
<!ELEMENT asamp   ( section+ )>
<!ELEMENT section ( title, para+ )>
<!ATTLIST section id ID #REQUIRED>
<!ELEMENT title   ( #PCDATA )>
<!ELEMENT para    ( #PCDATA | sectref )*>
<!ELEMENT sectref EMPTY>
<!ATTLIST sectref idref IDREF #REQUIRED>
]>
<asamp>
<section id="s1">
<title>Section One</title>
<para>First paragraph of section one.</para>
<para>Second paragraph of section one.</para>
</section>
<section id="s2">
<title>Section Two</title>
<para>First paragraph of section two.</para>
<para>Second paragraph of section two
with reference to <sectref idref="s1"/> embedded.</para>
</section>
</asamp>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Anchor elements (cont.)

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

With the desired result replacing the empty section reference elements with a hyperlink pointing to the section itself and the displayed content of the hyperlink being the title of the section to which it points:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Anchor Example</title>
</head>
<body>

<a name="s1">
<p style="font-size:15pt;color=red;text-align:center">Section One</p>
</a>
<p style="font-size:15pt;color=#004400">First paragraph of section
one.</p>
<p style="font-size:15pt;color=#004400">Second paragraph of section
one.</p>

<a name="s2">
<p style="font-size:15pt;color=red;text-align:center">Section Two</p>
</a>
<p style="font-size:15pt;color=#004400">First paragraph of section
two.</p>
<p style="font-size:15pt;color=#004400">Second paragraph of section two
with reference to <a href="#s1">Section One</a> embedded.</p>

</body>
</html>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Anchor elements (cont.)

Annex A - XML to HTML transformation
Section 3 - HTML markup generation techniques

The XSLT stylesheet `asamp.xsl` effects this transformation as follows, utilizing the XSLT `id()` function described in User XML identifier referencing (page 313):

```
<?xml version="1.0"?><!--asamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<!DOCTYPE xsl:stylesheet [ <!ENTITY dark-green "#004400"> ]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:output method="html"
         doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"/>

<xsl:template match="/">                    <!-- root rule -->
  <html>
    <head><title>Anchor Example</title></head>
    <body><xsl:apply-templates/></body></html></xsl:template>

<xsl:template match="section/title">
  <a name="{../@id}">
     <p style="font-size:15pt;color=red;text-align:center">
       <xsl:apply-templates/>
     </p></a></xsl:template>

<xsl:template match="para">
  <p style="font-size:15pt;color=#004400">
    <xsl:apply-templates/></p></xsl:template>

<xsl:template match="sectref">
  <a href="#{@idref}">
     <xsl:value-of select="id(@idref)/title"/></a></xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Annex B - XSL formatting semantics introduction

- Introduction - Formatting objectives
- Section 1 - Formatting model
- Section 2 - Formatting objects
- Section 3 - Example stylesheet with formatting constructs

Outcomes:

- awareness of the formatting objectives of the XSL development committee
- awareness of the formatting model

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Formatting objectives

The Extensible Stylesheet Language (XSL)

A catalogue of formatting objects and flow objects (each with properties controlling behavior) for rendering information to multiple media.

- addresses basic word-processing-level pagination
- semantic model for formatting
  - expressed in terms of which XSL concepts can be described
  - described as a vocabulary that can be serialized as XML markup

Sophisticated pagination and support for layout-driven documents

- DSSSL
  - Document Style Semantics and Specification Language ISO-10179
- W3C Common Formatting Model
  - effort initially based on CSS
- vocabulary accommodates both heritages
  - some constructs can be specified different ways with different names
  - writing-direction-independent (absolute) and writing-direction-dependent (writing mode relative) properties

Well-defined constructs

- express formatting intent
  - according to the XSL formatting model
- available to the stylesheet writer
  - for specification of a layout using the XSL formatting vocabulary
- managed and interpreted by the formatter
  - that process responsible for rendering
  - in response to a description of the layout

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Formatting objectives (cont.)

Effecting the formatting of XML with XSL formatting semantics

- transformation stylesheet
  - it is the stylesheet writer's responsibility to write an XSLT transformation of the XML source file into a result node tree composed entirely of formatting and flow objects using the XSL vocabulary
  - same architecture as when producing HTML from XML
  - an XSL-FO engine interprets an XSL-FO instance just as a browser interprets HTML
- semantics interpretation
  - an XSL processor implementing XSL formatting semantics recognizes the vocabulary and renders the result
- unlike CSS
  - the user's vocabulary is not supplemented with formatting properties

Intermediate result of rendering

- the XSL processor may, but need not, emit the result node tree as XML markup
  - formatting and flow objects are XML elements
  - properties are attribute/value pairs specified in the XML elements
- very useful for debugging stylesheets
- recall the XSL-FO engine incorporating XSLT (page 39)

This chapter briefly introduces concepts and basic constructs used in the XSL-FO 1.0 Recommendation, without going into the details of the vocabulary or markup required to support these concepts. The topic of formatting objects and their semantics and markup warrants an entire tutorial on its own and is thus separate from this tutorial.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of formatting model components

XSL incorporates ***most*** of the formatting objects and properties of CSS:

- 90% of the XSL properties are properties already defined in CSS
  - some old CSS properties now represent a shorthand definition of a collection of properties offering finer control
- new properties introduced for a model for pagination and layout
  - will be extended to page structures beyond the simple page models of the first version of the recommendation

Writing-direction-dependent properties

- some XSL constructs can be specified with adjectives that are writing mode relative
  - "`before`", "`after`", "`start`" and "`end`" vs. "`top`", "`bottom`", "`left`" and "`right`"
- stylesheets need not change to accommodate different writing directions
  - resulting rendering relative to writing direction

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of formatting model components (cont.)

The formatting model basis:

- rectangular areas of content
  - the content is flowed into the layout description
  - future extensions to the model may include non-rectangular areas
- spaces around and between content areas
  - the spaces make adjustments to the layout and do not have content

Area behavior:

- specified by the stylesheet writer
  - the stylesheet writer specifies areas and their traits using formatting objects and their properties
- managed by the formatter tasked with rendering the areas to the target device
  - areas may be filled explicitly by the stylesheet writer
    - traits explicitly specified in the stylesheet properties
  - areas may be filled implicitly by the formatter:
    - some traits implicitly derived
    - for example, if the stylesheet writer specifies a block that is filled with text, then the resulting line areas that comprise the block are synthesized by the formatter to accommodate the block
    - the formatter manages inheritance when creating areas and must derive certain properties when synthesizing areas not specified explicitly by the stylesheet writer
  - default values are included in the inheritance hierarchy

Important note:

- this design does not ensure page fidelity between two conforming implementations
- the declarative nature gives implementations leeway in certain aspects of rendering not specified by the stylesheet writer

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of formatting model components (cont.)

There is a hierarchy of rectangular areas managed by the stylesheet writer and the formatter:

- a container reference area:
  - contains block areas
  - may be placed at a specific position within a containing area
  - may be attached to the inside of any edge of a containing area
- a block area:
  - is filled (perpendicular to the writing direction) with line areas (that are in the writing direction) or nested block areas
  - stacks nested block areas within the containing area without the ability to span across a container boundary
- a line area:
  - generated within a block by the formatter
  - is filled with inline areas
- an inline area:
  - may contain other inline areas
  - may have complex content (e.g.: an inline mathematical expression)
  - at its lowest level contains a single glyph area or graphic area to be rendered

Each rectangular area has:

- a border surrounding the content of the area
- padding defining the open space between the inside of the border and the content

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of formatting model components (cont.)

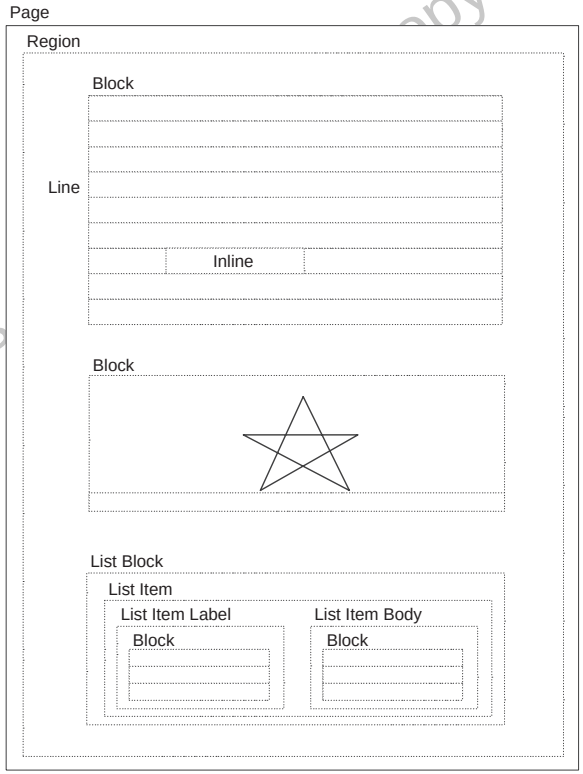
Space is managed by the stylesheet writer and the formatter:

- between blocks:
  - can be assigned preceding and following block areas to control placement
  - coalesces with adjacent display spaces according to resolution rules
- between inline areas:
  - is assigned preceding and following inline areas
  - coalesces with adjacent inline spaces according to resolution rules
- resolution rules include concepts of:
  - conditionality - whether or not a space is to exist
  - precedence - which aspect of space definition overrides others

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Summary of formatting model components (cont.)

A simple illustration of rectangular areas:



Laying out the areas requires knowledge of formatting objects, e.g.:

- blocks are as wide as their parents, thus can only stack above and below each other, never beside each other
- to get two blocks beside each other, one must introduce an object with pairs of adjacent areas that allow blocks as children, such as the list item object found within list block objects

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Formatting object vocabulary

Formatting and flow object vocabulary:

- identified by namespace URI and version
  - `http://www.w3.org/1999/XSL/Format`
  - 81 different objects defined
    - pagination and layout
    - block
    - inline
    - table
    - list
    - link and multi
    - out-of-line
    - other
- attributes specified in stylesheet control properties of objects
  - not all properties apply to all objects
  - 272 different properties defined
    - common accessibility properties
    - common absolute position properties
    - common aural properties
    - common border, padding and background properties
    - common font properties
    - common hyphenation properties
    - common keeps and breaks properties
    - common margin properties - block
    - common margin properties - inline
    - pagination and layout properties
    - table properties
    - character properties
    - rule and leader properties
    - page-related properties
    - float-related properties
    - number to string conversion properties
    - link properties
    - miscellaneous properties

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Example stylesheet with formatting constructs

The stylesheet `fmtsamp.xsl` illustrates an example of a stylesheet using the XSL formatting constructs for the result tree:

```
<?xml version="1.0"?><!--fmtsamp.xsl-->
<!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/1999/XSL/Format"
                version="1.0">
<xsl:output method="xml"/>

<xsl:param name="font-size" select="'20pt'"/>

<xsl:template match="/">   <!--put all on in one page sequence-->
  <root><layout-master-set>
    <simple-page-master master-name="crane"
                        margin-top=".5in" margin-bottom=".5in"
                        margin-left=".75in" margin-right=".75in"
                        page-width="8.5in" page-height="11in">
     <region-body region-name="crane-content"/>
    </simple-page-master></layout-master-set>
    <page-sequence master-reference="crane">
     <flow flow-name="crane-content" font-size="{$font-size}">
       <xsl:apply-templates/>
     </flow></page-sequence></root></xsl:template>

<xsl:template match="para">          <!--a standard paragraph-->
  <block space-before.optimum="{$font-size}">
    <xsl:apply-templates/></block></xsl:template>

<xsl:template match="emph">     <!--emphasize some information-->
  <inline font-weight="bold">
    <xsl:apply-templates/></inline></xsl:template>

</xsl:stylesheet>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Example stylesheet with formatting constructs (cont.)

When processing the source file `fmtsamp.xml`:

```
<?xml version="1.0"?>
<doc>
<para>First paragraph.</para>
<para>Second para <emph>with emphasis</emph> embedded.</para>
<para>Last paragraph.</para>
</doc>
```

Produces the result file `fmtsamp.fo`:

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns="http://www.w3.org/1999/XSL/Format">
<layout-master-set><simple-page-master master-name="crane"
margin-top=".5in" margin-bottom=".5in" margin-left=".75in"
margin-right=".75in" page-width="8.5in" page-height="11in">
<region-body region-name="crane-content"/>
</simple-page-master></layout-master-set>
<page-sequence master-reference="crane">
<flow flow-name="crane-content" font-size="20pt">
<block space-before.optimum="20pt">First paragraph.</block>
<block space-before.optimum="20pt">Second para <inline
font-weight="bold">with emphasis</inline> embedded.</block>
<block space-before.optimum="20pt">Last paragraph.</block>
</flow></page-sequence></root>
```

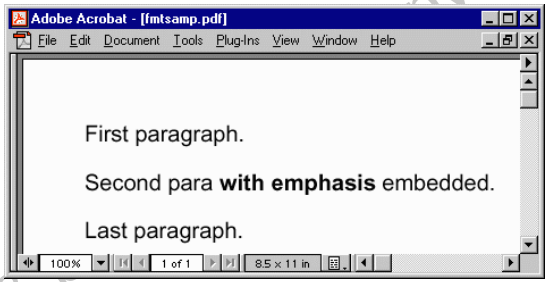

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Example stylesheet with formatting constructs (cont.)

Annex B - XSL formatting semantics introduction
Section 3 - Example stylesheet with formatting constructs

When rendered to PDF using FOP `http://xml.apache.org/fop/`:



Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Annex C - Instruction, function and grammar summaries

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Quick summaries

This annex lists alphabetized references to the components of the specifications. Each entry notes the chapter in this book where the construct is primarily described.

The specifications are rigorous references to all of the facilities and functions:

XSLT 1.0/XPath 1.0:

- `http://www.w3.org/TR/1999/REC-xslt-19991116`
- `http://www.w3.org/TR/1999/REC-xpath-19991116`

XSLT 2.0/XPath 2.0/XQuery 1.0

- `http://www.w3.org/TR/2007/REC-xslt20-20070123/`
- `http://www.w3.org/TR/2007/REC-xpath20-20070123/`
- `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/`
- `http://www.w3.org/TR/2007/REC-xpath-functions-20070123/`
- `http://www.w3.org/TR/2007/REC-xslt-xquery-serialization-20070123/`
- `http://www.w3.org/TR/2007/REC-xquery-20070123/`
- `http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/`
- `http://www.w3.org/TR/2007/REC-xqueryx-20070123`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT 1.0 element summary

Annex C - Instruction, function and grammar summaries
Section 1 - Vocabulary, functions and grammars XSLT 1.0 and XPath 1.0

All elements in the XSLT vocabulary in alphabetical order follow. Note that the Kleene operators '?', '*' and '+' (respectively zero or one, zero or more, and one or more) are used to denote the cardinality of attributes and contained constructs. The content model operators ',' and '|' (respectively sequence and alternation) are also used. The brace brackets '{' and '}' denote the use of an attribute value template. This information is mechanically derived from the XSLT 1.0 Recommendation.

`apply-imports` (instruction) - Imported stylesheets (page 250)
- XSLT 1.0 5.6 Overriding Template Rules
-
```
<xsl:apply-imports/>
```

`apply-templates` (instruction) - Repositioning using "push" (page 158)
- XSLT 1.0 5.4 Applying Template Rules
-
```
<xsl:apply-templates mode="qname"?
                     select="node-set-expression"?>
  (<xsl:sort>|<xsl:with-param>)*
</xsl:apply-templates>
```

`attribute` (instruction) - Constructing attribute nodes (page 361)
- XSLT 1.0 7.1 Creating Elements and Attributes
-
```
<xsl:attribute name="qname|{string-expression}"
               namespace="uri-reference|{string-expression}"?>
  template
</xsl:attribute>
```

`attribute-set` (top level element) - Constructing attribute nodes (page 362)
- XSLT 1.0 7.1 Creating Elements and Attributes
-
```
<xsl:attribute-set name="qname"
                   use-attribute-sets="qnames"?>
  <xsl:attribute>*
</xsl:attribute-set>
```

`call-template` (instruction) - Named templates (page 232)
- XSLT 1.0 6 Named Templates
-
```
<xsl:call-template name="qname">
  <xsl:with-param>*
</xsl:call-template>
```

`choose` (instruction) - "If - Else If - Else" conditionality (page 140)
- XSLT 1.0 9.2 Conditional Processing with `xsl:choose`
-
```
<xsl:choose>
  (<xsl:when>+,<xsl:otherwise>?)
</xsl:choose>
```

`comment` (instruction) - Constructing annotation nodes (page 366)
- XSLT 1.0 7.4 Creating Comments
-
```
<xsl:comment>
  template
</xsl:comment>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`copy` (instruction) - Copying source tree nodes to the result tree (page 379)
- XSLT 1.0 7.5 Copying
-
```
<xsl:copy use-attribute-sets="qnames"?>
  template
</xsl:copy>
```

`copy-of` (instruction) - Copying source tree nodes to the result tree (page 379)
- XSLT 1.0 11.3 Using Values of Variables and Parameters with `xsl:copy-of`
-
```
<xsl:copy-of select="expression"/>
```

`decimal-format` (top level element) - Decimal formatting in XSLT (page 295)
- XSLT 1.0 12.3 Number Formatting
-
```
<xsl:decimal-format decimal-separator="char"?
                    digit="char"?
                    grouping-separator="char"?
                    infinity="string"?
                    minus-sign="char"?
                    name="qname"?
                    NaN="string"?
                    pattern-separator="char"?
                    per-mille="char"?
                    percent="char"?
                    zero-digit="char"?/>
```

`element` (instruction) - Constructing element nodes (page 364)
- XSLT 1.0 7.1 Creating Elements and Attributes
-
```
<xsl:element name="qname|{string-expression}"
             namespace="uri-reference|{string-expression}"?
             use-attribute-sets="qnames"?>
  template
</xsl:element>
```

`fallback` (instruction) - Extension mechanisms (page 256)
- XSLT 1.0 15 Fallback
-
```
<xsl:fallback>
  template
</xsl:fallback>
```

`for-each` (instruction) - Repositioning using "pull" (page 155)
- XSLT 1.0 8 Repetition
-
```
<xsl:for-each select="node-set-expression">
  (<xsl:sort>*,template)
</xsl:for-each>
```

`if` (instruction) - "If - Then" conditionality (page 139)
- XSLT 1.0 9.1 Conditional Processing with `xsl:if`
-
```
<xsl:if test="boolean-expression">
  template
</xsl:if>
```

`import` - Imported stylesheets (page 246)
- XSLT 1.0 2.6 Combining Stylesheets
-
```
<xsl:import href="uri-reference"/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

include (top level element) - Included stylesheets (page 245)

- XSLT 1.0 2.6 Combining Stylesheets
-
```
<xsl:include href="uri-reference"/>
```

key (top level element) - XSLT key node referencing (page 320)

- XSLT 1.0 12.2 Keys
-
```
<xsl:key match="pattern"
         name="qname"
         use="expression"/>
```

message (instruction) - Communication facilities in XSLT (page 206)

- XSLT 1.0 13 Messages
-
```
<xsl:message terminate="yes|no"?>
  template
</xsl:message>
```

namespace-alias (top level element) - Namespace protection (page 187)

- XSLT 1.0 7.1 Creating Elements and Attributes
-
```
<xsl:namespace-alias result-prefix="prefix|#default"
                     stylesheet-prefix="prefix|#default"/>
```

number (instruction) - Source tree numbering (page 391)

- XSLT 1.0 7.7 Numbering
-
```
<xsl:number count="pattern"?
            format="string|{string-expression}"?
            from="pattern"?
            grouping-separator="char|{string-expression}"?
            grouping-size="number|{string-expression}"?
            lang="nmtoken|{string-expression}"?

letter-value="alphabetic|traditional|{string-expression}"?
            level="single|multiple|any"?
            value="number-expression"?/>
```

otherwise - "If - Else If - Else" conditionality (page 140)

- XSLT 1.0 9.2 Conditional Processing with xsl:choose
-
```
<xsl:otherwise>
  template
</xsl:otherwise>
```

output (top level element) - Serializing the result tree (page 191)

- XSLT 1.0 16 Output
-
```
<xsl:output cdata-section-elements="qnames"?
            doctype-public="string"?
            doctype-system="string"?
            encoding="string"?
            indent="yes|no"?
            media-type="string"?
            method="xml|html|text|qname-but-not-ncname"?
            omit-xml-declaration="yes|no"?
            standalone="yes|no"?
            version="nmtoken"?/>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

param (top level element) - Variable and parameter binding (page 225)

- XSLT 1.0 11 Variables and Parameters
-
```
<xsl:param name="qname"
           select="expression"?>
  template
</xsl:param>
```

preserve-space (top level element) - White-space-only text nodes (page 88)

- XSLT 1.0 3.4 Whitespace Stripping
-
```
<xsl:preserve-space elements="tokens"/>
```

processing-instruction (instruction) - Constructing annotation nodes (page 366)

- XSLT 1.0 7.3 Creating Processing Instructions
-
```
<xsl:processing-instruction name="ncname|{string-expression}">
  template
</xsl:processing-instruction>
```

sort - The sort instruction (page 407)

- XSLT 1.0 10 Sorting
-
```
<xsl:sort case-order="upper-first|lower-first|{string-expression}"?

data-type="text|number|qname-but-not-ncname|{string-expression}"?
          lang="nmtoken|{string-expression}"?
          order="ascending|descending|{string-expression}"?
          select="string-expression"?/>
```

strip-space (top level element) - White-space-only text nodes (page 88)

- XSLT 1.0 3.4 Whitespace Stripping
-
```
<xsl:strip-space elements="tokens"/>
```

stylesheet - The stylesheet document/container element (page 176)

- XSLT 1.0 2.2 Stylesheet Element
-
```
<xsl:stylesheet version="number"
                exclude-result-prefixes="tokens"?
                extension-element-prefixes="tokens"?
                id="id"?>
  (<xsl:import>*,top-level-elements)
</xsl:stylesheet>
```

template (top level element) - Repositioning using "push" (page 159)

- XSLT 1.0 5.3 Defining Template Rules
-
```
<xsl:template match="pattern"?
              mode="qname"?
              name="qname"?
              priority="number"?>
  (<xsl:param>*,template)
</xsl:template>
```

text (instruction) - Constructing text nodes (page 371)

- XSLT 1.0 7.2 Creating Text
-
```
<xsl:text disable-output-escaping="yes|no"?>
  #PCDATA
</xsl:text>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`transform` - The stylesheet document/container element (page 176)

- XSLT 1.0 2.2 Stylesheet Element
- ```
  <xsl:transform version="number"
                 exclude-result-prefixes="tokens"?
                 extension-element-prefixes="tokens"?
                 id="id"?>
    (<xsl:import>*,top-level-elements)
  </xsl:transform>
  ```

`value-of` (instruction) - Constructing result text (page 150)

- XSLT 1.0 7.6 Computing Generated Text
- ```
  <xsl:value-of select="string-expression"
                disable-output-escaping="yes|no"?/>
  ```

`variable` (top level element) - Variable and parameter binding (page 224)

- XSLT 1.0 11 Variables and Parameters
- ```
  <xsl:variable name="qname"
                select="expression"?>
    template
  </xsl:variable>
  ```

`when` - "If - Else If - Else" conditionality (page 140)

- XSLT 1.0 9.2 Conditional Processing with `xsl:choose`
- ```
  <xsl:when test="boolean-expression">
    template
  </xsl:when>
  ```

`with-param` - Named templates (page 233)

- XSLT 1.0 11.6 Passing Parameters to Templates
- ```
  <xsl:with-param name="qname"
                  select="expression"?>
    template
  </xsl:with-param>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 1.0 and XSLT 1.0 function summary

Annex C - Instruction, function and grammar summaries
Section 1 - Vocabulary, functions and grammars XSLT 1.0 and XPath 1.0

All functions of both XPath 1.0 and XSLT 2.0 in alphabetical order follow. This information is mechanically derived from the XPath 1.0 and XSLT 1.0 Recommendations.

`boolean` - Calculating values using Boolean functions (page 331)

- XPath 1.0 4.3 Boolean Functions
- `boolean boolean( object )`

`ceiling` - Calculating values using number functions (page 285)

- XPath 1.0 4.4 Number Functions
- `number ceiling( number )`

`concat` - Calculating values using string functions (page 290)

- XPath 1.0 4.2 String Functions
- `string concat( string, string, string* )`

`contains` - Calculating values using string functions (page 292)

- XPath 1.0 4.2 String Functions
- `boolean contains( string, string )`

`count` - Sequence operator and functions (page 325)

- XPath 1.0 4.1 Node Set Functions
- `number count( node-set )`

`current` - Current node referencing in XSLT (page 324)

- XSLT 1.0 12.4 Miscellaneous Additional Functions
- `node-set current( )`

`document` - Document referencing in XSLT (page 262)

- XSLT 1.0 12.1 Multiple Source Documents
- `node-set document( object, node-set? )`

`element-available` - Extension mechanisms (page 255)

- XSLT 1.0 15 Fallback
- `boolean element-available( string )`

`false` - Calculating values using Boolean functions (page 331)

- XPath 1.0 4.3 Boolean Functions
- `boolean false( )`

`floor` - Calculating values using number functions (page 285)

- XPath 1.0 4.4 Number Functions
- `number floor( number )`

`format-number` - Decimal formatting in XSLT (page 294)

- XSLT 1.0 12.3 Number Formatting
- `string format-number( number, string, string? )`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
  ```

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Practical Transformation Using XSLT and XPath

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XPath 1.0 grammar productions

Annex C - Instruction, function and grammar summaries
Section 1 - Vocabulary, functions and grammars XSLT 1.0 and XPath 1.0

Location Paths (2)

```
[1] LocationPath ::= RelativeLocationPath[3]
                   | AbsoluteLocationPath[2]
[2] AbsoluteLocationPath ::= '/' RelativeLocationPath[3]?
                           | AbbreviatedAbsoluteLocationPath[10]
[3] RelativeLocationPath ::= Step[4]
                           | RelativeLocationPath[3] '/' Step[4]
                           | AbbreviatedRelativeLocationPath[11]
```

Location Steps (2.1)

```
[4] Step ::= AxisSpecifier[5] NodeTest[7] Predicate[8]*
           | AbbreviatedStep[12]
[5] AxisSpecifier ::= AxisName[6] '::'
                    | AbbreviatedAxisSpecifier[13]
```

Axes (2.2)

```
[6] AxisName ::= 'ancestor'
               | 'ancestor-or-self'
               | 'attribute'
               | 'child'
               | 'descendant'
               | 'descendant-or-self'
               | 'following'
               | 'following-sibling'
               | 'namespace'
               | 'parent'
               | 'preceding'
               | 'preceding-sibling'
               | 'self'
```

Node Tests (2.3)

```
[7] NodeTest ::= NameTest[37]
               | NodeType[38] '(' ')'
               | 'processing-instruction' '(' Literal[29] ')'
```

Predicates (2.4)

```
[8] Predicate ::= '[' PredicateExpr[9] ']'
[9] PredicateExpr ::= Expr[14]
```

Abbreviated Syntax (2.5)

```
[10] AbbreviatedAbsoluteLocationPath ::= '//' RelativeLocationPath[3]
[11] AbbreviatedRelativeLocationPath ::= RelativeLocationPath[3] '//' Step[4]
[12] AbbreviatedStep ::= '.'
                       | '..'
[13] AbbreviatedAxisSpecifier ::= '@'?
```

Expressions (3)

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Basics (3.1)

```
[14] Expr ::= OrExpr[21]
[15] PrimaryExpr ::= VariableReference[36]
                   | '(' Expr[14] ')'
                   | Literal[29]
                   | Number[30]
                   | FunctionCall[16]
```

Function Calls (3.2)

```
[16] FunctionCall ::= FunctionName[35] '(' ( Argument[17] ( ',' Argument[17]
                     )* )? ')'
[17] Argument ::= Expr[14]
```

Node-sets (3.3)

```
[18] UnionExpr ::= PathExpr[19]
                 | UnionExpr[18] '|' PathExpr[19]
[19] PathExpr ::= LocationPath[1]
                | FilterExpr[20]
                | FilterExpr[20] '/' RelativeLocationPath[3]
                | FilterExpr[20] '//' RelativeLocationPath[3]
[20] FilterExpr ::= PrimaryExpr[15]
                  | FilterExpr[20] Predicate[8]
```

Booleans (3.4)

```
[21] OrExpr ::= AndExpr[22]
              | OrExpr[21] 'or' AndExpr[22]
[22] AndExpr ::= EqualityExpr[23]
               | AndExpr[22] 'and' EqualityExpr[23]
[23] EqualityExpr ::= RelationalExpr[24]
                    | EqualityExpr[23] '=' RelationalExpr[24]
                    | EqualityExpr[23] '!=' RelationalExpr[24]
[24] RelationalExpr ::= AdditiveExpr[25]
                      | RelationalExpr[24] '<' AdditiveExpr[25]
                      | RelationalExpr[24] '>' AdditiveExpr[25]
                      | RelationalExpr[24] '<=' AdditiveExpr[25]
                      | RelationalExpr[24] '>=' AdditiveExpr[25]
```

Numbers (3.5)

```
[25] AdditiveExpr ::= MultiplicativeExpr[26]
                    | AdditiveExpr[25] '+' MultiplicativeExpr[26]
                    | AdditiveExpr[25] '-' MultiplicativeExpr[26]
[26] MultiplicativeExpr ::= UnaryExpr[27]
                          | MultiplicativeExpr[26] MultiplyOperator[34]
                          UnaryExpr[27]
                          | MultiplicativeExpr[26] 'div' UnaryExpr[27]
                          | MultiplicativeExpr[26] 'mod' UnaryExpr[27]
[27] UnaryExpr ::= UnionExpr[18]
                 | '-' UnaryExpr[27]
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Lexical Structure (3.7)

```
[28] ExprToken ::= '(' | ')' | '[' | ']' | '.' | '..' | '@' | ',' | '::'
                 | NameTest[37]
                 | NodeType[38]
                 | Operator[32]
                 | FunctionName[35]
                 | AxisName[6]
                 | Literal[29]
                 | Number[30]
                 | VariableReference[36]
[29] Literal ::= '"' [^"]* '"'
               | "'" [^']* "'"
[30] Number ::= Digits[31] ('.' Digits[31]?)?
              | '.' Digits[31]
[31] Digits ::= [0-9]+
[32] Operator ::= OperatorName[33]
                | MultiplyOperator[34]
                | '/' | '//' | '|' | '+' | '-' | '=' | '!=' | '<' | '<='
                | '>' | '>='
[33] OperatorName ::= 'and' | 'or' | 'mod' | 'div'
[34] MultiplyOperator ::= '*'
[35] FunctionName ::= QName[XML-Names-6] - NodeType[38]
[36] VariableReference ::= '$' QName[XML-Names-6]
[37] NameTest ::= '*'
                | NCName[XML-Names-4] ':' '*'
                | QName[XML-Names-6]
[38] NodeType ::= 'comment'
                | 'text'
                | 'processing-instruction'
                | 'node'
[39] ExprWhitespace ::= S[XML-3]
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT 1.0 grammar productions

Annex C - Instruction, function and grammar summaries
Section 1 - Vocabulary, functions and grammars XSLT 1.0 and XPath 1.0

Template Rules (5)

Patterns (5.2)

```
[1] Pattern ::= LocationPathPattern[2]
              | Pattern[1] '|' LocationPathPattern[2]
[2] LocationPathPattern ::= '/' RelativePathPattern[4]?
                          | IdKeyPattern[3] (('/' | '//')
                          RelativePathPattern[4])?
                          | '//'? RelativePathPattern[4]
[3] IdKeyPattern ::= 'id' '(' Literal[XPath-1.0-29] ')'
                   | 'key' '(' Literal[XPath-1.0-29] ',' Literal[XPath-1.0-29]
                   ')'
[4] RelativePathPattern ::= StepPattern[5]
                          | RelativePathPattern[4] '/' StepPattern[5]
                          | RelativePathPattern[4] '//' StepPattern[5]
[5] StepPattern ::= ChildOrAttributeAxisSpecifier[6] NodeTest[XPath-1.0-7]
                  Predicate[XPath-1.0-8]*
[6] ChildOrAttributeAxisSpecifier ::= AbbreviatedAxisSpecifier[XPath-1.0-13]
                                    | ('child' | 'attribute') '::'
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT 2.0 element summary

Annex C - Instruction, function and grammar summaries
Section 2 - Vocabulary, functions and grammars XSLT 2.0 and XPath 2.0

All elements in the XSLT vocabulary in alphabetical order follow. Note that the Kleene operators '?', '*' and '+' (respectively zero or one, zero or more, and one or more) are used to denote the cardinality of attributes and contained constructs. The content model operators ',' and '|' (respectively sequence and alternation) are also used. The brace brackets '{' and '}' denote the use of an attribute value template. This information is mechanically derived from the XSLT 1.0 Recommendation.

`analyze-string` - String analysis in XSLT 2.0 (page 305)

- XSLT 2.0 - 15.1 The xsl:analyze-string instruction

```
- <!-- Category: instruction -->
  <xsl:analyze-string
    select = expression
    regex = { string }
    flags? = { string }>
   <!-- Content: (xsl:matching-substring?, xsl:non-matching-substring?,
   xsl:fallback*) -->
  </xsl:analyze-string>
```

`apply-imports` - Imported stylesheets (page 250)

- XSLT 2.0 - 6.7 Overriding Template Rules

```
- <!-- Category: instruction -->
  <xsl:apply-imports>
    <!-- Content: xsl:with-param* -->
  </xsl:apply-imports>
```

`apply-templates` - Repositioning using "push" (page 158)

- XSLT 2.0 - 6.3 Applying Template Rules

```
- <!-- Category: instruction -->
  <xsl:apply-templates
    select? = expression
    mode? = token>
    <!-- Content: (xsl:sort | xsl:with-param)* -->
  </xsl:apply-templates>
```

`attribute` - Constructing attribute nodes (page 361)

- XSLT 2.0 - 11.3 Creating Attribute Nodes Using xsl:attribute

```
- <!-- Category: instruction -->
  <xsl:attribute
    name = { qname }
    namespace? = { uri-reference }
    select? = expression
    separator? = { string }
    type? = qname
    validation? = "strict" | "lax" | "preserve" | "strip">
    <!-- Content: sequence-constructor -->
  </xsl:attribute>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`attribute-set` - Constructing attribute nodes (page 362)

- XSLT 2.0 - 10.2 Named Attribute Sets

```
- <!-- Category: declaration -->
  <xsl:attribute-set
    name = qname
    use-attribute-sets? = qnames>
    <!-- Content: xsl:attribute* -->
  </xsl:attribute-set>
```

`call-template` - Named templates (page 232)

- XSLT 2.0 - 10.1 Named Templates

```
- <!-- Category: instruction -->
  <xsl:call-template
    name = qname>
    <!-- Content: xsl:with-param* -->
  </xsl:call-template>
```

`character-map` - Character maps (page 199)

- XSLT 2.0 - 20.1 Character Maps

```
- <!-- Category: declaration -->
  <xsl:character-map
    name = qname
    use-character-maps? = qnames>
    <!-- Content: (xsl:output-character*) -->
  </xsl:character-map>
```

`choose` - "If - Else If - Else" conditionality (page 140)

- XSLT 2.0 - 8.2 Conditional Processing with xsl:choose

```
- <!-- Category: instruction -->
  <xsl:choose>
    <!-- Content: (xsl:when+, xsl:otherwise?) -->
  </xsl:choose>
```

`comment` - Constructing annotation nodes (page 366)

- XSLT 2.0 - 11.8 Creating Comments

```
- <!-- Category: instruction -->
  <xsl:comment
    select? = expression>
    <!-- Content: sequence-constructor -->
  </xsl:comment>
```

`copy` - Copying source tree nodes to the result tree (page 379)

- XSLT 2.0 - 11.9.1 Shallow Copy

```
- <!-- Category: instruction -->
  <xsl:copy
    copy-namespaces? = "yes" | "no"
    inherit-namespaces? = "yes" | "no"
    use-attribute-sets? = qnames
    type? = qname
    validation? = "strict" | "lax" | "preserve" | "strip">
    <!-- Content: sequence-constructor -->
  </xsl:copy>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`copy-of` - Copying source tree nodes to the result tree (page 379)

- XSLT 2.0 - 11.9.2 Deep Copy

```
- <!-- Category: instruction -->
  <xsl:copy-of
    select = expression
    copy-namespaces? = "yes" | "no"
    type? = qname
    validation? = "strict" | "lax" | "preserve" | "strip" />
```

`decimal-format` - Decimal formatting in XSLT (page 295)

- XSLT 2.0 - 16.4.1 Defining a Decimal Format

```
- <!-- Category: declaration -->
  <xsl:decimal-format
    name? = qname
    decimal-separator? = char
    grouping-separator? = char
    infinity? = string
    minus-sign? = char
    NaN? = string
    percent? = char
    per-mille? = char
    zero-digit? = char
    digit? = char
    pattern-separator? = char />
```

`document` - Constructing document nodes (page 369)

- XSLT 2.0 - 11.5 Creating Document Nodes

```
- <!-- Category: instruction -->
  <xsl:document
    validation? = "strict" | "lax" | "preserve" | "strip"
    type? = qname>
    <!-- Content: sequence-constructor -->
  </xsl:document>
```

`element` - Constructing element nodes (page 364)

- XSLT 2.0 - 11.2 Creating Element Nodes Using xsl:element

```
- <!-- Category: instruction -->
  <xsl:element
    name = { qname }
    namespace? = { uri-reference }
    inherit-namespaces? = "yes" | "no"
    use-attribute-sets? = qnames
    type? = qname
    validation? = "strict" | "lax" | "preserve" | "strip">
    <!-- Content: sequence-constructor -->
  </xsl:element>
```

`fallback` - Extension mechanisms (page 256)

- XSLT 2.0 - 18.2.3 Fallback

```
- <!-- Category: instruction -->
  <xsl:fallback>
    <!-- Content: sequence-constructor -->
  </xsl:fallback>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`for-each` - Repositioning using "pull" (page 155)

- XSLT 2.0 - 7 Repetition

```
- <!-- Category: instruction -->
  <xsl:for-each
    select = expression>
    <!-- Content: (xsl:sort*, sequence-constructor) -->
  </xsl:for-each>
```

`for-each-group` - Built-in grouping facilities in XSLT 2.0 (page 434)

- XSLT 2.0 - 14.3 The xsl:for-each-group Element

```
- <!-- Category: instruction -->
  <xsl:for-each-group
    select = expression
    group-by? = expression
    group-adjacent? = expression
    group-starting-with? = pattern
    group-ending-with? = pattern
    collation? = { uri }>
    <!-- Content: (xsl:sort*, sequence-constructor) -->
  </xsl:for-each-group>
```

`function` - User-defined functions (page 240)

- XSLT 2.0 - 10.3 Stylesheet Functions

```
- <!-- Category: declaration -->
  <xsl:function
    name = qname
    as? = sequence-type
    override? = "yes" | "no">
    <!-- Content: (xsl:param*, sequence-constructor) -->
  </xsl:function>
```

`if` - "If - Then" conditionality (page 139)

- XSLT 2.0 - 8.1 Conditional Processing with xsl:if

```
- <!-- Category: instruction -->
  <xsl:if
    test = expression>
    <!-- Content: sequence-constructor -->
  </xsl:if>
```

`import` - Imported stylesheets (page 246)

- XSLT 2.0 - 3.10.3 Stylesheet Import

```
- <!-- Category: declaration -->
  <xsl:import
    href = uri-reference />
```

`import-schema` - Importing schema definitions (page 185)

- XSLT 2.0 - 3.14 Importing Schema Components

```
- <!-- Category: declaration -->
  <xsl:import-schema
    namespace? = uri-reference
    schema-location? = uri-reference>
    <!-- Content: xs:schema? -->
  </xsl:import-schema>
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`include` - Included stylesheets (page 245)

- XSLT 2.0 - 3.10.2 Stylesheet Inclusion
- ```
  <!-- Category: declaration -->
  <xsl:include
    href = uri-reference />
  ```

`key` - XSLT key node referencing (page 320)

- XSLT 2.0 - 16.3.1 The xsl:key Declaration
- ```
  <!-- Category: declaration -->
  <xsl:key
    name = qname
    match = pattern
    use? = expression
    collation? = uri>
    <!-- Content: sequence-constructor -->
  </xsl:key>
  ```

`matching-substring` - String analysis in XSLT 2.0 (page 305)

- XSLT 2.0 - 15.1 The xsl:analyze-string instruction
- ```
  <xsl:matching-substring>
    <!-- Content: sequence-constructor -->
  </xsl:matching-substring>
  ```

`message` - Communication facilities in XSLT (page 206)

- XSLT 2.0 - 17 Messages
- ```
  <!-- Category: instruction -->
  <xsl:message
    select? = expression
    terminate? = { "yes" | "no" }>
    <!-- Content: sequence-constructor -->
  </xsl:message>
  ```

`namespace` - Constructing namespace nodes (page 368)

- XSLT 2.0 - 11.7 Creating Namespace Nodes
- ```
  <!-- Category: instruction -->
  <xsl:namespace
    name = { ncname }
    select? = expression>
    <!-- Content: sequence-constructor -->
  </xsl:namespace>
  ```

`namespace-alias` - Namespace protection (page 187)

- XSLT 2.0 - 11.1.4 Namespace Aliasing
- ```
  <!-- Category: declaration -->
  <xsl:namespace-alias
    stylesheet-prefix = prefix | "#default"
    result-prefix = prefix | "#default" />
  ```

`next-match` - Imported stylesheets (page 250)

- XSLT 2.0 - 6.7 Overriding Template Rules
- ```
  <!-- Category: instruction -->
  <xsl:next-match>
    <!-- Content: (xsl:with-param | xsl:fallback)* -->
  </xsl:next-match>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`non-matching-substring` - String analysis in XSLT 2.0 (page 305)

- XSLT 2.0 - 15.1 The xsl:analyze-string instruction
- ```
  <xsl:non-matching-substring>
    <!-- Content: sequence-constructor -->
  </xsl:non-matching-substring>
  ```

`number` - Source tree numbering (page 391)

- XSLT 2.0 - 12 Numbering
- ```
  <!-- Category: instruction -->
  <xsl:number
    value? = expression
    select? = expression
    level? = "single" | "multiple" | "any"
    count? = pattern
    from? = pattern
    format? = { string }
    lang? = { nmtoken }
    letter-value? = { "alphabetic" | "traditional" }
    ordinal? = { string }
    grouping-separator? = { char }
    grouping-size? = { number
  } />
  ```

`otherwise` - "If - Else If - Else" conditionality (page 140)

- XSLT 2.0 - 8.2 Conditional Processing with xsl:choose
- ```
  <xsl:otherwise>
    <!-- Content: sequence-constructor -->
  </xsl:otherwise>
  ```

`output` - Serializing the result tree (page 191)

- XSLT 2.0 - 20 Serialization
- ```
  <!-- Category: declaration -->
  <xsl:output
    name? = qname
    method? = "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname
    byte-order-mark? = "yes" | "no"
    cdata-section-elements? = qnames
    doctype-public? = string
    doctype-system? = string
    encoding? = string
    escape-uri-attributes? = "yes" | "no"
    include-content-type? = "yes" | "no"
    indent? = "yes" | "no"
    media-type? = string
    normalization-form? = "NFC" | "NFD" | "NFKC" | "NFKD" |
  "fully-normalized" | "none" | nmtoken
    omit-xml-declaration? = "yes" | "no"
    standalone? = "yes" | "no" | "omit"
    undeclare-prefixes? = "yes" | "no"
    use-character-maps? = qnames
    version? = nmtoken />
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
  ```

`output-character` - Character maps (page 199)

- XSLT 2.0 - 20.1 Character Maps
- ```
  <xsl:output-character
    character = char
    string = string />
  ```

`param` - Variable and parameter binding (page 225)

- XSLT 2.0 - 9.2 Parameters
- ```
  <!-- Category: declaration -->
  <xsl:param
    name = qname
    select? = expression
    as? = sequence-type
    required? = "yes" | "no"
    tunnel? = "yes" | "no">
    <!-- Content: sequence-constructor -->
  </xsl:param>
  ```

`perform-sort` - Sorting sequences (page 406)

- XSLT 2.0 - 13.2 Creating a Sorted Sequence
- ```
  <!-- Category: instruction -->
  <xsl:perform-sort
    select? = expression>
    <!-- Content: (xsl:sort+, sequence-constructor) -->
  </xsl:perform-sort>
  ```

`preserve-space` - White-space-only text nodes (page 88)

- XSLT 2.0 - 4.4 Stripping Whitespace from a Source Tree
- ```
  <!-- Category: declaration -->
  <xsl:preserve-space
    elements = tokens />
  ```

`processing-instruction` - Constructing annotation nodes (page 366)

- XSLT 2.0 - 11.6 Creating Processing Instructions
- ```
  <!-- Category: instruction -->
  <xsl:processing-instruction
    name = { ncname }
    select? = expression>
    <!-- Content: sequence-constructor -->
  </xsl:processing-instruction>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`result-document` - Multiple result trees (page 201)

- XSLT 2.0 - 19.1 Creating Final Result Trees
- ```
  <!-- Category: instruction -->
  <xsl:result-document
    format? = { qname }
    href? = { uri-reference }
    validation? = "strict" | "lax" | "preserve" | "strip"
    type? = qname
    method? = { "xml" | "html" | "xhtml" | "text" | qname-but-not-ncname
   }
    byte-order-mark? = { "yes" | "no" }
    cdata-section-elements? = { qnames }
    doctype-public? = { string }
    doctype-system? = { string }
    encoding? = { string }
    escape-uri-attributes? = { "yes" | "no" }
    include-content-type? = { "yes" | "no" }
    indent? = { "yes" | "no" }
    media-type? = { string }
    normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD"
  | "fully-normalized" | "none" | nmtoken }
    omit-xml-declaration? = { "yes" | "no" }
    standalone? = { "yes" | "no" | "omit" }
    undeclare-prefixes? = { "yes" | "no" }
    use-character-maps? = qnames
    output-version? = { nmtoken }>
    <!-- Content: sequence-constructor -->
  </xsl:result-document>
  ```

`sequence` - Named templates (page 236)

- XSLT 2.0 - 11.10 Constructing Sequences
- ```
  <!-- Category: instruction -->
  <xsl:sequence
    select = expression>
    <!-- Content: xsl:fallback* -->
  </xsl:sequence>
  ```

`sort` - The sort instruction (page 407)

- XSLT 2.0 - 13.1 The xsl:sort Element
- ```
  <xsl:sort
    select? = expression
    lang? = { nmtoken }
    order? = { "ascending" | "descending" }
    collation? = { uri }
    stable? = { "yes" | "no" }
    case-order? = { "upper-first" | "lower-first" }
    data-type? = { "text" | "number" | qname-but-not-ncname }>
    <!-- Content: sequence-constructor -->
  </xsl:sort>
  ```

`strip-space` - White-space-only text nodes (page 88)

- XSLT 2.0 - 4.4 Stripping Whitespace from a Source Tree
- ```
  <!-- Category: declaration -->
  <xsl:strip-space
    elements = tokens />
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
  ```

`stylesheet` - The stylesheet document/container element (page 176)

- XSLT 2.0 - 3.6 Stylesheet Element
- ```
  <xsl:stylesheet
    id? = id
    extension-element-prefixes? = tokens
    exclude-result-prefixes? = tokens
    version = number
    xpath-default-namespace? = uri
    default-validation? = "preserve" | "strip"
    default-collation? = uri-list
    input-type-annotations? = "preserve" | "strip" | "unspecified">
    <!-- Content: (xsl:import*, other-declarations) -->
  </xsl:stylesheet>
  ```

`template` - Repositioning using "push" (page 159)

- XSLT 2.0 - 6.1 Defining Templates
- ```
  <!-- Category: declaration -->
  <xsl:template
    match? = pattern
    name? = qname
    priority? = number
    mode? = tokens
    as? = sequence-type>
    <!-- Content: (xsl:param*, sequence-constructor) -->
  </xsl:template>
  ```

`text` - Constructing text nodes (page 371)

- XSLT 2.0 - 11.4.2 Creating Text Nodes Using xsl:text
- ```
  <!-- Category: instruction -->
  <xsl:text
    [disable-output-escaping]?
  = "yes" | "no">
    <!-- Content: #PCDATA -->
  </xsl:text>
  ```

`transform` - The stylesheet document/container element (page 176)

- XSLT 2.0 - 3.6 Stylesheet Element
- ```
  <xsl:transform
    id? = id
    extension-element-prefixes? = tokens
    exclude-result-prefixes? = tokens
    version = number
    xpath-default-namespace? = uri
    default-validation? = "preserve" | "strip"
    default-collation? = uri-list
    input-type-annotations? = "preserve" | "strip" | "unspecified">
    <!-- Content: (xsl:import*, other-declarations) -->
  </xsl:transform>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`value-of` - Constructing result text (page 150)

- XSLT 2.0 - 11.4.3 Generating Text with xsl:value-of
- ```
  <!-- Category: instruction -->
  <xsl:value-of
    select? = expression
    separator? = { string }
    [disable-output-escaping]?
  = "yes" | "no">
    <!-- Content: sequence-constructor -->
  </xsl:value-of>
  ```

`variable` - Variable and parameter binding (page 224)

- XSLT 2.0 - 9.1 Variables
- ```
  <!-- Category: declaration -->
  <!-- Category: instruction -->
  <xsl:variable
    name = qname
    select? = expression
    as? = sequence-type>
    <!-- Content: sequence-constructor -->
  </xsl:variable>
  ```

`when` - "If - Else If - Else" conditionality (page 140)

- XSLT 2.0 - 8.2 Conditional Processing with xsl:choose
- ```
  <xsl:when
    test = expression>
    <!-- Content: sequence-constructor -->
  </xsl:when>
  ```

`with-param` - Named templates (page 233)

- XSLT 2.0 - 10.1.1 Passing Parameters to Templates
- ```
  <xsl:with-param
    name = qname
    select? = expression
    as? = sequence-type
    tunnel? = "yes" | "no">
    <!-- Content: sequence-constructor -->
  </xsl:with-param>
  ```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XPath 2.0 and XSLT 2.0 function summary

Annex C - Instruction, function and grammar summaries
Section 2 - Vocabulary, functions and grammars XSLT 2.0 and XPath 2.0

All functions of both XPath 2.0 and XSLT 2.0 in alphabetical order follow. This information is mechanically derived from the XPath 2.0 Functions and XSLT 2.0 Recommendations.

`abs` - Calculating values using number functions (page 285)
- XPath 2.0 - 6.4 Functions on Numeric Values
- *`numeric`* `abs(` *`numeric`* `)`

`adjust-date-to-timezone` - Date and time functions and operators (page 341)
- XPath 2.0 - 10.7 Timezone Adjustment Functions on Dates and Time Values
- *`xs:date`* `adjust-date-to-timezone(` *`xs:date`* `)`

  `xs:date` `adjust-date-to-timezone(` *`xs:date`*`,` *`xs:dayTimeDuration`* `)`

`adjust-dateTime-to-timezone` - Date and time functions and operators (page 341)
- XPath 2.0 - 10.7 Timezone Adjustment Functions on Dates and Time Values
- *`xs:dateTime`* `adjust-dateTime-to-timezone(` *`xs:dateTime`* `)`

  `xs:dateTime` `adjust-dateTime-to-timezone(` *`xs:dateTime`*`,` *`xs:dayTimeDuration`* `)`

`adjust-time-to-timezone` - Date and time functions and operators (page 341)
- XPath 2.0 - 10.7 Timezone Adjustment Functions on Dates and Time Values
- *`xs:time`* `adjust-time-to-timezone(` *`xs:time`* `)`

  `xs:time` `adjust-time-to-timezone(` *`xs:time`*`,` *`xs:dayTimeDuration`* `)`

`avg` - Sequence operator and functions (page 330)
- XPath 2.0 - 15.4 Aggregate Functions
- *`xs:anyAtomicType`* `avg(` *`xs:anyAtomicType*`* `)`

`base-uri` - Calculating values using node-set-related expression functions (page 309)
- XPath 2.0 - 2 Accessors
- *`xs:anyURI`* `base-uri(  )`

  `xs:anyURI` `base-uri(` *`node()`* `)`

`boolean` - Calculating values using Boolean functions (page 331)
- XPath 2.0 - 15.1 General Functions and Operators on Sequences
- *`xs:boolean`* `boolean(` *`item()*`* `)`

`ceiling` - Calculating values using number functions (page 285)
- XPath 2.0 - 6.4 Functions on Numeric Values
- *`numeric`* `ceiling(` *`numeric`* `)`

`codepoint-equal` - Calculating values using string functions (page 290)
- XPath 2.0 - 7.3 Equality and Comparison of Strings
- *`xs:boolean`* `codepoint-equal(` *`xs:string`*`,` *`xs:string`* `)`

`codepoints-to-string` - Calculating values using string functions (page 290)
- XPath 2.0 - 7.2 Functions to Assemble and Disassemble Strings
- *`xs:string`* `codepoints-to-string(` *`xs:integer*`* `)`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

`collection` - Document referencing in XPath 2 (page 261)
- XPath 2.0 - 15.5 Functions and Operators that Generate Sequences
- *`node()*`* `collection(  )`

  `node()*` `collection(` *`xs:string`* `)`

`compare` - Calculating values using string functions (page 290)
- XPath 2.0 - 7.3 Equality and Comparison of Strings
- *`xs:integer`* `compare(` *`xs:string`*`,` *`xs:string`* `)`

  `xs:integer` `compare(` *`xs:string`*`,` *`xs:string`*`,` *`xs:string`* `)`

`concat` - Calculating values using string functions (page 290)
- XPath 2.0 - 7.4 Functions on String Values
- *`xs:string`* `concat(` *`xs:anyAtomicType`*`,` *`xs:anyAtomicType`*`,  )`

`contains` - Calculating values using string functions (page 292)
- XPath 2.0 - 7.5 Functions Based on Substring Matching
- *`xs:boolean`* `contains(` *`xs:string`*`,` *`xs:string`* `)`

  `xs:boolean` `contains(` *`xs:string`*`,` *`xs:string`*`,` *`xs:string`* `)`

`count` - Sequence operator and functions (page 325)
- XPath 2.0 - 15.4 Aggregate Functions
- *`xs:integer`* `count(` *`item()*`* `)`

`current` - Current node referencing in XSLT (page 324)
- XSLT 2.0 - 16.6.1 current
- *`item()`* `current(  )`

`current-date` - Date and time functions and operators (page 341)
- XPath 2.0 - 16 Context Functions
- *`xs:date`* `current-date(  )`

`current-dateTime` - Date and time functions and operators (page 341)
- XPath 2.0 - 16 Context Functions
- *`xs:dateTime`* `current-dateTime(  )`

`current-group` - Built-in grouping facilities in XSLT 2.0 (page 435)
- XSLT 2.0 - 14.1 The Current Group
- *`item()*`* `current-group(  )`

`current-grouping-key` - Built-in grouping facilities in XSLT 2.0 (page 435)
- XSLT 2.0 - 14.2 The Current Grouping Key
- *`xs:anyAtomicType?`* `current-grouping-key(  )`

`current-time` - Date and time functions and operators (page 341)
- XPath 2.0 - 16 Context Functions
- *`xs:time`* `current-time(  )`

`data` - Calculating values using node-set-related expression functions (page 309)
- XPath 2.0 - 2 Accessors
- *`xs:anyAtomicType*`* `data(` *`item()*`* `)`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Practical Transformation Using XSLT and XPath

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

XPath 2.0 grammar productions

Annex C - Instruction, function and grammar summaries
Section 2 - Vocabulary, functions and grammars XSLT 2.0 and XPath 2.0

Expressions (3)
[1] `XPath` ::= `Expr`[2]
[2] `Expr` ::= `ExprSingle`[3] ("," `ExprSingle`[3])*
[3] `ExprSingle` ::= `ForExpr`[4]| `QuantifiedExpr`[6]| `IfExpr`[7]| `OrExpr`[8]

For Expressions (3.7)
[4] `ForExpr` ::= `SimpleForClause`[5] "return" `ExprSingle`[3]
[5] `SimpleForClause` ::= "for" "$" `VarName`[45] "in" `ExprSingle`[3] ("," "$" `VarName`[45] "in" `ExprSingle`[3])*

Quantified Expressions (3.9)
[6] `QuantifiedExpr` ::= ("some" | "every") "$" `VarName`[45] "in" `ExprSingle`[3] ("," "$" `VarName`[45] "in" `ExprSingle`[3])* "satisfies" `ExprSingle`[3]

Conditional Expressions (3.8)
[7] `IfExpr` ::= "if" "(" `Expr`[2] ")" "then" `ExprSingle`[3] "else" `ExprSingle`[3]

Logical Expressions (3.6)
[8] `OrExpr` ::= `AndExpr`[9] ("or" `AndExpr`[9])*
[9] `AndExpr` ::= `ComparisonExpr`[10] ("and" `ComparisonExpr`[10])*

Comparison Expressions (3.5)
[10] `ComparisonExpr` ::= `RangeExpr`[11] ((`ValueComp`[23]| `GeneralComp`[22]| `NodeComp`[24]) `RangeExpr`[11])?

Constructing Sequences (3.3.1)
[11] `RangeExpr` ::= `AdditiveExpr`[12] ("to" `AdditiveExpr`[12])?

Arithmetic Expressions (3.4)
[12] `AdditiveExpr` ::= `MultiplicativeExpr`[13] (("+" | "-") `MultiplicativeExpr`[13])*
[13] `MultiplicativeExpr` ::= `UnionExpr`[14] (("*" | "div" | "idiv" | "mod") `UnionExpr`[14])*

Combining Node Sequences (3.3.3)
[14] `UnionExpr` ::= `IntersectExceptExpr`[15] (("union" | "|") `IntersectExceptExpr`[15])*
[15] `IntersectExceptExpr` ::= `InstanceofExpr`[16] (("intersect" | "except") `InstanceofExpr`[16])*

Instance Of (3.10.1)
[16] `InstanceofExpr` ::= `TreatExpr`[17] ("instance" "of" `SequenceType`[50])?

Treat (3.10.5)
[17] `TreatExpr` ::= `CastableExpr`[18] ("treat" "as" `SequenceType`[50])?

Castable (3.10.3)
[18] `CastableExpr` ::= `CastExpr`[19] ("castable" "as" `SingleType`[49])?

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Cast (3.10.2)
[19] `CastExpr` ::= `UnaryExpr`[20] ("cast" "as" `SingleType`[49])?

Arithmetic Expressions (3.4)
[20] `UnaryExpr` ::= ("-" | "+")* `ValueExpr`[21]
[21] `ValueExpr` ::= `PathExpr`[25]

Comparison Expressions (3.5)
[22] `GeneralComp` ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[23] `ValueComp` ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[24] `NodeComp` ::= "is" | "<<" | ">>"

Path Expressions (3.2)
[25] `PathExpr` ::= ("/" `RelativePathExpr`[26]?)| ("//" `RelativePathExpr`[26])| `RelativePathExpr`[26]
[26] `RelativePathExpr` ::= `StepExpr`[27] (("/" | "//") `StepExpr`[27])*

Steps (3.2.1)
[27] `StepExpr` ::= `FilterExpr`[38] | `AxisStep`[28]
[28] `AxisStep` ::= (`ReverseStep`[32] | `ForwardStep`[29]) `PredicateList`[39]
[29] `ForwardStep` ::= (`ForwardAxis`[30] `NodeTest`[35]) | `AbbrevForwardStep`[31]

Axes (3.2.1.1)
[30] `ForwardAxis` ::= ("child" "::")| ("descendant" "::")| ("attribute" "::")| ("self" "::")| ("descendant-or-self" "::")| ("following-sibling" "::")| ("following" "::")| ("namespace" "::")

Abbreviated Syntax (3.2.4)
[31] `AbbrevForwardStep` ::= "@"? `NodeTest`[35]

Steps (3.2.1)
[32] `ReverseStep` ::= (`ReverseAxis`[33] `NodeTest`[35]) | `AbbrevReverseStep`[34]

Axes (3.2.1.1)
[33] `ReverseAxis` ::= ("parent" "::")| ("ancestor" "::")| ("preceding-sibling" "::")| ("preceding" "::")| ("ancestor-or-self" "::")

Abbreviated Syntax (3.2.4)
[34] `AbbrevReverseStep` ::= ".."

Node Tests (3.2.1.2)
[35] `NodeTest` ::= `KindTest`[54] | `NameTest`[36]
[36] `NameTest` ::= `QName`[78] | `Wildcard`[37]
[37] `Wildcard` ::= "*"| (`NCName`[79] ":" "*")| ("*" ":" `NCName`[79])

Filter Expressions (3.3.2)
[38] `FilterExpr` ::= `PrimaryExpr`[41] `PredicateList`[39]

Steps (3.2.1)
[39] `PredicateList` ::= `Predicate`[40]*

Predicates (3.2.2)
[40] `Predicate` ::= "[" `Expr`[2] "]"

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Primary Expressions (3.1)

```
[41] PrimaryExpr ::= Literal[42] | VarRef[44] | ParenthesizedExpr[46] |
                     ContextItemExpr[47] | FunctionCall[48]
```

Literals (3.1.1)

```
[42] Literal ::= NumericLiteral[43] | StringLiteral[74]
[43] NumericLiteral ::= IntegerLiteral[71] | DecimalLiteral[72] |
                        DoubleLiteral[73]
```

Variable References (3.1.2)

```
[44] VarRef ::= "$" VarName[45]
[45] VarName ::= QName[78]
```

Parenthesized Expressions (3.1.3)

```
[46] ParenthesizedExpr ::= "(" Expr[2]? ")"
```

Context Item Expression (3.1.4)

```
[47] ContextItemExpr ::= "."
```

Function Calls (3.1.5)

```
[48] FunctionCall ::= QName[78] "(" (ExprSingle[3] ("," ExprSingle[3])*)? ")"
```

Cast (3.10.2)

```
[49] SingleType ::= AtomicType[53] "?"?
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

SequenceType Syntax (2.5.3)

```
[50] SequenceType ::= ("empty-sequence" "(" ")")| (ItemType[52]
                      OccurrenceIndicator[51]?)
[51] OccurrenceIndicator ::= "?" | "*" | "+"
[52] ItemType ::= KindTest[54] | ("item" "(" ")") | AtomicType[53]
[53] AtomicType ::= QName[78]
[54] KindTest ::= DocumentTest[56]| ElementTest[64]| AttributeTest[60]|
                  SchemaElementTest[66]| SchemaAttributeTest[62]| PITest[59]|
                  CommentTest[58]| TextTest[57]| AnyKindTest[55]
[55] AnyKindTest ::= "node" "(" ")"
[56] DocumentTest ::= "document-node" "(" (ElementTest[64] |
                      SchemaElementTest[66])? ")"
[57] TextTest ::= "text" "(" ")"
[58] CommentTest ::= "comment" "(" ")"
[59] PITest ::= "processing-instruction" "(" (NCName[79] | StringLiteral[74])?
                ")"
[60] AttributeTest ::= "attribute" "(" (AttribNameOrWildcard[61] ","
                       TypeName[70])?)? ")"
[61] AttribNameOrWildcard ::= AttributeName[68] | "*"
[62] SchemaAttributeTest ::= "schema-attribute" "(" AttributeDeclaration[63]
                             ")"
[63] AttributeDeclaration ::= AttributeName[68]
[64] ElementTest ::= "element" "(" (ElementNameOrWildcard[65] ","
                     TypeName[70] "?"?)?)? ")"
[65] ElementNameOrWildcard ::= ElementName[69] | "*"
[66] SchemaElementTest ::= "schema-element" "(" ElementDeclaration[67] ")"
[67] ElementDeclaration ::= ElementName[69]
[68] AttributeName ::= QName[78]
[69] ElementName ::= QName[78]
[70] TypeName ::= QName[78]
```

Literals (3.1.1)

```
[71] IntegerLiteral ::= Digits[81]
[72] DecimalLiteral ::= ("." Digits[81]) | (Digits[81] "." [0-9]*)
[73] DoubleLiteral ::= (("." Digits[81]) | (Digits[81] ("." [0-9]*)?)) [eE]
                       [+-]? Digits[81]
[74] StringLiteral ::= ('"' (EscapeQuot[75] | [^"])* '"') | ("'"
                       (EscapeApos[76] | [^'])* "'")
[75] EscapeQuot ::= '""'
[76] EscapeApos ::= "''"
```

Comments (2.6)

```
[77] Comment ::= "(:" (CommentContents[82] | Comment[77])* ":)"
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Terminal Symbols (1.2.1)

[78] `QName` ::= `[http://www.w3.org/TR/REC-xml-names/#NT-QName]`[XML-Names-7]

[79] `NCName` ::= `[http://www.w3.org/TR/REC-xml-names/#NT-NCName]`[XML-Names-4]

[80] `Char` ::= `[http://www.w3.org/TR/REC-xml#NT-Char]`[XML-2]

Literals (3.1.1)

[81] `Digits` ::= `[0-9]+`

Comments (2.6)

[82] `CommentContents` ::= `(Char`[80]`+ - (Char* ('(:' | ':)') Char*))`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLT 2.0 grammar productions

Annex C - Instruction, function and grammar summaries
Section 2 - Vocabulary, functions and grammars XSLT 2.0 and XPath 2.0

[1] `Pattern` ::= `PathPattern`[2]
| `Pattern`[1] `'|'` `PathPattern`[2]

[2] `PathPattern` ::= `RelativePathPattern`[3]
| `'/'` `RelativePathPattern`[3]?
| `'//'` `RelativePathPattern`[3]
| `IdKeyPattern`[6] `(('/' | '//')` `RelativePathPattern`[3]`)?`

[3] `RelativePathPattern` ::= `PatternStep`[4] `(('/' | '//')`
`RelativePathPattern`[3]`)?`

[4] `PatternStep` ::= `PatternAxis`[5]`?` `NodeTest`[XPath-35] `PredicateList`[XPath-39]

[5] `PatternAxis` ::= `('child' '::' | 'attribute' '::' | '@')`

[6] `IdKeyPattern` ::= `'id' '('` `IdValue`[7] `')'`
| `'key' '('` `StringLiteral`[XPath-74] `','` `KeyValue`[8] `')'`

[7] `IdValue` ::= `StringLiteral`[XPath-74] | `VarRef`[XPath-44]

[8] `KeyValue` ::= `Literal`[XPath-42] | `VarRef`[XPath-44]

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Annex D - Tool questions

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sample questions for vendors

Answers to the following questions may prove useful when trying to better understand a product offering from a vendor. The specific questions are grouped under topical questions. This by no means makes up a complete list of questions as you may have your own criteria to add, nonetheless, they do cover aspects of XSLT and XQuery that may impact on the stylesheets and transformation specifications you write.

- how is the product identified?
  - what is the name of the processor in product literature?
  - what value is returned by the system properties?
    - recall Communication facilities in XSLT (page 207)
  - what version of specification is supported?
    - returned by the `xsl:version` system property in XSLT
  - to which email address or URL are questions forwarded for more information in general?
  - to which email address or URL are questions forwarded for more information specific to the answers to these technical questions?
- what output serialization methods are supported for the result node tree?
  - XML?
  - HTML?
  - text?
  - XHTML?
  - XSL formatting and flow objects?
    - in what ways are the formatting objects interpreted (direct to screen? HTML? PostScript? PDF? TeX? etc.)?
  - other non-XML text-oriented methods different than the standard text method (e.g. NXML by XT)?
    - what are the semantics and vocabulary for each such environment?
  - other custom serialization methods?
    - what are the semantics and vocabulary for each such environment?
  - what customization is available to implement one's own interpretation of result tree semantics?
    - is there access to the result tree as either a DOM tree or SAX events?
    - does such access still oblige serialization to an external file?

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Sample questions for vendors (cont.)

Annex D - Tool questions

- how does the processor differ from the W3C working drafts or recommendations?
  - upon which dated W3C documents describing the specifications is the software based?
  - which constructs or functions are not implemented at all?
  - which constructs or functions are implemented differently than in the W3C description?
  - what namespace URI values are used for those available constructs or functions described differently or not described in W3C version?
  - is the W3C recommended stylesheet association technique implemented for the direct processing XML instances?
    - if so, can it be selectively engaged and disengaged?
- are any extension functions or extension elements implemented?
  - what is the recognized extension namespace and the utility of the extension functions and elements implemented?
    - is there an extension function for the conversion of a result tree fragment to a node-set?
    - T are there any built-in extension functions or extension elements for the writing of templates to an output URL?
  - can additional extension functions or extension elements (beyond those supplied by the vendor) be added by the user?
    - how so?
  - are any extensions defined by `exslt.org` supported?

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Sample questions for vendors (cont.)

Annex D - Tool questions

- how are particular facilities implemented?
  - what is the implementation in the processor of `indent="yes"` for `<xsl:output>`?
  - is a method provided for defining top-level `<xsl:param>` constructs at invocation time?
  - how is the `<xsl:message>` construct implemented?
  - which UCS/Unicode format tokens are supported for `<xsl:number>`?
  - which `lang=` values are supported for `<xsl:sort>`?
  - T2 what is the URI syntax for data projection for input?
  - T2 which collations are supported for string comparison?
  - T2 is the processor schema-aware?
  - T2 what are the details of the collection URI syntax?
- how are errors reported or gracefully handled?
  - regarding template conflict resolution?
  - regarding improper content of result tree nodes (e.g. comments, processing instructions)?
  - regarding invocation of unimplemented functions or features?
  - regarding any other areas?
  - can fatal error reporting (e.g. template conflict resolution or other errors) be selectively turned on to diagnose stylesheets targeted for use with other XSLT processors that fail on an error?

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Sample questions for vendors (cont.)

Annex D - Tool questions

- what are the details of the implementation and invocation of the processor?
  - how are user values passed in to the transformation?
  - which hardware/operating system platforms support the processor?
  - which character sets are supported for the input file encoding and output serialization?
  - what is the XML processor used within the XSLT processor?
    - does the XML processor support minimally declared internal declaration subsets with only attribute list declarations of `ID`-typed attributes?
    - does the XML processor support XML Inclusions (Xinclude)?
    - does the XML processor support catalogues for public identifiers?
    - does the XML processor validate the source file?
      - can this be turned on and off?
  - can the processor be embedded in other applications?
    - can the processor be configured as a servlet in a web server?
    - is there access to the result tree as either a DOM tree or SAX events?
  - is the source code of the processor available?
    - in what language is the processor written?
  - for Windows-based environments:
    - can the processor be invoked from the MSDOS command-line box?
    - can the processor be invoked from a GUI interface?
    - what other methods of invocation can be triggered (DLL, RPC, etc.)?
    - can error messages be explicitly redirected to a file using an invocation parameter (since, for example, Windows-95 does not allow for redirection of the standard error port to a file)?
  - does the processor take advantage of parallelism when executing the stylesheet, or is the stylesheet always processed serially?
  - does the processor implement tail recursion for called named templates?
  - does the processor implement lazy evaluation for XPath location path expression evaluation?

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLStyle™

Annex D - Tool questions
Section 1 - XSLStyle™

An XSLT stylesheet embedded documentation methodology

- an XSLT stylesheet is an XML document
- one can embed a documentation vocabulary in an XSLT stylesheet through standard XML namespace techniques
- a stylesheet for stylesheets renders the embedded documentation to HTML and CSS
- freely downloadable environment from the Crane Softwrights Ltd. web site as a developer resource

Can invoke as a separate stylesheet or through embedded association

- see `http://www.w3.org/TR/xml-stylesheet/`
- recall Stylesheet association (page 34)

Embedded documentation uses a Crane namespace for structure

- DocBook for content
  - see `http://www.docbook.org/` for details
- DITA for content
  - see `http://dita.xml.org/` for details
- XHTML for content

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLStyle™ (cont.)

Annex D - Tool questions
Section 1 - XSLStyle™

Enforces "stylesheet writing rules" on the writer of the stylesheet

- adds rigor to the stylesheet
- e.g. all top-level constructs must be separately documented
- e.g. all parameters of all templates and functions must be separately documented
- e.g. all parameters of all templates and functions must have declared types
- e.g. all named top-level constructs must be namespace qualified
- many other rules

Resulting HTML report is similar to an enhanced Javadoc report

- encompasses the complete import tree
- alphabetized index of all named top-level constructs
- deficiencies report
  - fully hyperlinked content
  - one could institute a development rule of not allowing the check-in of a stylesheet until the library is fully documented according to the writing rules

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLStyle™ (cont.)

Annex D - Tool questions
Section 1 - XSLStyle™

An example fragment:

```
<?xml-stylesheet type="text/xsl" href="xslstyle-docbook.xsl"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.CraneSoftwrights.com/ns/xslstyle"
                xmlns:i="internal-namespace"
                exclude-result-prefixes="xs i"
                version="1.0">

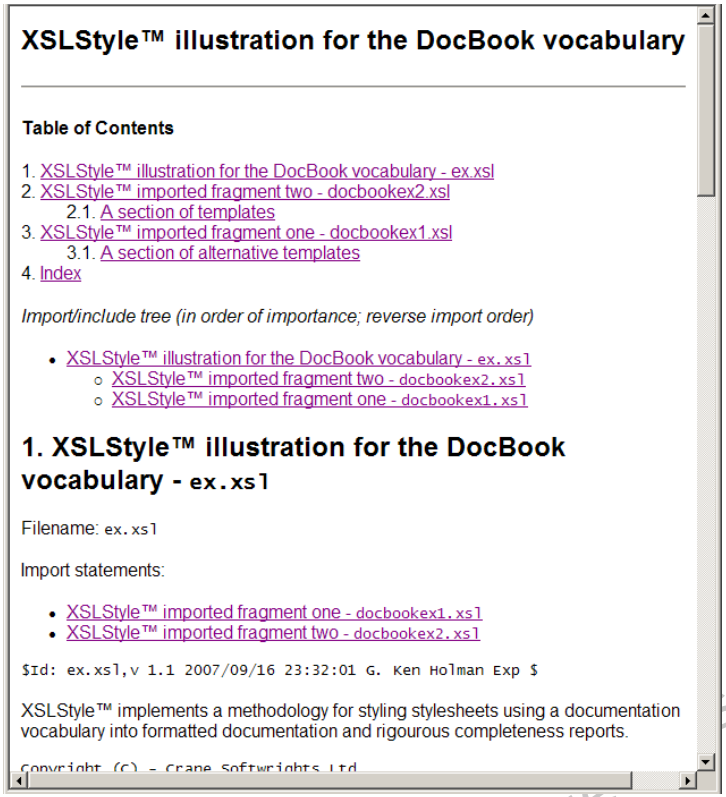
<!--an example import-->
<xsl:import href="docbookex1.xsl"/>
<!--another example import-->
<xsl:import href="docbookex2.xsl"/>

<xs:doc info="$Id: ex.xsl,v 1.1 2007/09/16 23:32:01 G. Ken Holman Exp
$"
  filename="ex.xsl" global-ns="xs" internal-ns="i" vocabulary="DocBook">
  <xs:title>XSLStyle&#x2122; illustration for the DocBook
vocabulary</xs:title>
  <para>
    XSLStyle&#x2122; implements a methodology for styling stylesheets
    using a documentation vocabulary into formatted documentation and
    rigourous completeness reports.
  </para>
  <programlisting>
Copyright (C) - Crane Softwrights Ltd.
...
<xs:template>
  <para>The formatting of a single entry in the import tree.</para>
  <xs:param name="href">
    <para>The URI used to access the module for this entry.</para>
  </xs:param>
</xs:template>
<xsl:template name="i:format-tree-entry">
  <xsl:param name="href"/>
  <listitem>
...
```


Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

XSLStyle™ (cont.)

Annex D - Tool questions
Section 1 - XSLStyle™



XSLStyle™ illustration for the DocBook vocabulary

Table of Contents

1. XSLStyle™ illustration for the DocBook vocabulary - ex.xsl
2. XSLStyle™ imported fragment two - docbookex2.xsl
   2.1. A section of templates
3. XSLStyle™ imported fragment one - docbookex1.xsl
   3.1. A section of alternative templates
4. Index

Import/include tree (in order of importance; reverse import order)

- XSLStyle™ illustration for the DocBook vocabulary - ex.xsl
  - XSLStyle™ imported fragment two - docbookex2.xsl
  - XSLStyle™ imported fragment one - docbookex1.xsl

1. XSLStyle™ illustration for the DocBook vocabulary - ex.xsl

Filename: ex.xsl

Import statements:

- XSLStyle™ imported fragment one - docbookex1.xsl
- XSLStyle™ imported fragment two - docbookex2.xsl

```
$Id: ex.xsl,v 1.1 2007/09/16 23:32:01 G. Ken Holman Exp $
```

XSLStyle™ implements a methodology for styling stylesheets using a documentation vocabulary into formatted documentation and rigourous completeness reports.

Copyright (C) - Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Where to go from here?

Conclusion - Practical Transformation Using XSLT and XPath

The work on XSL, XQuery, XSLT and XPath continues:

- all are full W3C Recommendations undergoing designs for new features
- long list of future feature considerations already being examined for new releases of the technology
- new products are continually being announced
- feedback is necessary from users like you!
  - use the XSL mail lists to contribute:
    - `http://www.mulberrytech.com/xsl/xsl-list/`
    - `http://groups.yahoo.com/group/XSL-FO`
    - `http://lists.w3.org/Archives/Public/www-xsl-fo/`
  - contact the W3C with comments about the XSLT/XPath/XQuery specifications:
    - `http://www.w3.org/XML/2005/04/qt-bugzilla`
    - `mailto:public-qt-comments@w3.org`

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Colophon

Conclusion - Practical Transformation Using XSLT and XPath

These materials were produced using structured information technologies as follows:

- authored source materials
  - content in numerous XML files maintained as external general entities for a complete prose book that can be made into a subset for training
    - specification of applicability of constructs for each configuration
      - 45- and 90-minute lecture, half-, full-, two- and three-day lecture and hands-on instruction, and book (prose) configurations
    - an XSLT transformation creates the subset of effective constructs from applying applicability to the complete file
    - content from other presentations/tutorials included semantically (not syntactically) during construct assembly
  - customized appearance engaged with marked sections and both parameter and general entities
    - different host company logos and venue and date marginalia
    - changing a single external parameter entity to a key file includes suite of files for given appearance
- accessible rendition in HTML
  - an XSLT stylesheet produces a collection of HTML files using Saxon for multiple file output
  - mono-spaced fonts and list-depth notation conventions assist the comprehension of the material when using screen-reader software
- printed handout deliverables
  - an XSLT stylesheet produces an instance of XSL formatting objects (XSL-FO) for rendering
  - XPDF `http://www.foolabs.com/xpdf` extracts raw text from PDF files for the back-of-the-book index methodology published as a free resource by Crane Softwrights Ltd.
  - XEP by RenderX `http://www.renderx.com` produces PostScript from XSL-FO
  - GhostScript `http://www.GhostScript.com` produces PDF from PostScript
  - the iText `http://itext.sf.net` PDF manipulation library for Java is used for page imposition by a custom Python `http://www.python.org` program running under the Jython `http://www.jython.org` environment

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Obtaining a copy of the comprehensive tutorial

Conclusion - Practical Transformation Using XSLT and XPath

This comprehensive tutorial on XSLT and XPath is available for subscription purchase and free preview download:

- "Practical Transformation Using XSLT and XPath (XSL Transformations and the XML Path Language)" Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
  - the free download preview excerpt of the publication indicates the number of pages for each topic
- the cost of purchase includes all future updates to the materials with email notification
  - the materials are updated after new releases of the W3C specifications
  - the materials are updated after incorporating comments gleaned during presentations and from feedback from customers
- available in PDF
  - formatted as 1-up or 2-up book pages per imaged page
  - dimensions in either US-letter or A4 page sizes
  - available as either single sided or double sided
- accessible rendition available for use with screen readers
- free preview download includes full text of first two chapters and two useful annexes
- site-wide and world-wide staff licenses (one-time fee) are available

See `http://www.CraneSoftwrights.com/links/trn-20110211.htm` for more details.

Feedback

- the unorthodox style has been well-accepted by customers as an efficient learning presentation
- feedback from customers is important to improve or repair the content for future editions
- please send suggestions or comments (positive or negative) to `info@CraneSoftwrights.com`

US Government employee purchase

- US Government employees (not contractors) are entitled to obtain their personal prepaid copies at no charge from a government intranet location
- visit the Crane web site for details

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.

Practical Transformation Using XSLT and XPath (XSL Transformations and the XML Path Language)

Crane Softwrights Ltd.
http://www.CraneSoftwrights.com

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Table of contents
Indexed by slide number

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Index

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Copyright © Crane Softwrights Ltd.
Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5
Information subject to restrictive legend on first page.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Practical Transformation Using XSLT and XPath

Copyright © Crane Softwrights Ltd.

Fourteenth Edition - 2011-02-11 - ISBN 978-1-894049-24-5

Information subject to restrictive legend on first page.