CSP2103-4102: Markup Languages

Lecture 6: Computational XSLT

AUSTRALIA ECU EDITH COWAN UNIVERSITY

Learning Outcomes

- Learn how to number nodes
- Apply XPath functions such as count() and sum()
- Create formulas using mathematical operators
- Work with text nodes and white space
- Create variables and parameters
- Create named and recursive templates
- Work with multiple style sheets
- Learn how to use extension functions and elements

Numbering Nodes

- Number nodes using:
  - <xsl:number>
  - position()
- Using position():
  - Nodes are numbered by position in result document
  - Can be used in conditional statements

Using <XSL:number>

- Nodes are numbered according to position in source document
- Attributes:
  - value=expression: any XPath expression that evaluates to a number (i.e. position())
  - count=pattern: specifies which nodes to count
  - level=type: tree level for nodes to count; can be any, single, or multiple

Using <XSL:number>

- Attributes:
  - from=pattern: pattern indicates where numbering should restart
  - format=pattern: pattern indicates number format
  - grouping-size, grouping-separator: indicate how digits are grouped and separator character

Working With Xpath Functions

- Used to calculate numerical values or manipulate text strings
- Numerical functions:

XPATH NUMERIC FUNCTIONS

FUNCTION	DESCRIPTION
ceiling(*number*)	Rounds *number* up to the nearest integer
count(*node_set*)	Counts the number of nodes in *node_set*
floor(*number*)	Rounds *number* down to the nearest integer
last(*node_set*)	Returns the index of the last node in *node_set*
position()	Returns the position of the context node within the processed node set
round(*number*)	Rounds *number* to the nearest integer
sum(*node_set*)	Calculates the sum of the values of *node_set*

Xpath Text Functions

XPATH TEXT STRING FUNCTIONS

FUNCTION	DESCRIPTION
concat(*string1*, *string2*, *string3*, ...)	Combines *string1*, *string2*, *string3*, ... into a single text string
contains(*string1*, *string2*)	Returns the value true if *string1* contains the text string, *string2* and false if otherwise
starts-with(*string1*, *string2*)	Returns the value true if *string1* begins with the characters defined in *string2* and false if otherwise
string(*object*)	Converts *object* to a text string. If *object* is not specified, the string() function returns the string value of the context node.
string-length(*string*)	Returns the length of *string*. If *string* is not specified, the string-length() function returns the length of the string value of the context node.
substring(*string*, *start*, *length*)	Returns a substring from *string*, starting with the character in the *start* position and continuing for *length* characters. If no length is specified, the substring goes to the end of the original text string.
substring-after(*string1*, *string2*)	Returns a substring of *string1* that occurs after the characters defined in *string2*
substring-before(*string1*, *string2*)	Returns a substring of *string1* that occurs before the characters defined in *string2*

Working With Mathematical Operators

- Six operators:

XPATH MATHEMATICAL OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE
+	Adds two numbers together	3 + 5
–	Subtracts one number from another	5 – 3
*	Multiplies two numbers together	5 * 3
div	Divides one number by another	15 divided by 3
mod	Provides the remainder after performing a division of one number by another	15 mod 3
–	Negates a single number	–2

Formatting Numbers

- XPath function format-number
- Syntax: format-number(value, format)
- Example: format-number(56823.847, “#,##0.00”) displays 56,823.85

Number Format Symbols

NUMBER FORMAT SYMBOLS

SYMBOL	DESCRIPTION
#	Placeholder that displays an optional number of digits in the formatted number and is usually used as the leftmost symbol in the number format
0	Placeholder that displays required digits in the formatted number
.	Separates the integer digits from the fractional digits
,	Separates groups of digits in the number
;	Separates the pattern for positive numbers from the pattern for negative numbers
–	Shows the location of the minus symbol for negative numbers
%	Multiplies the number by 100 and displays the number as a percentage
‰	Multiplies the number by 1000 and displays the number as a per-mille value

<XSL:decimal-format>

- Holds decimal formatting information
- Controls separator characters such as . and ,
- Can be named or default if un-named
- Named decimal format passed as argument to format-number

<XSL:decimal-format> Attributes

ATTRIBUTES OF THE <XSL:DECIMAL-FORMAT> ELEMENT

ATTRIBUTE	DESCRIPTION
name	Name of the decimal format. If you omit a name, the numbering scheme becomes the default format for the document.
decimal-separator	Character used to separate the integer and fractional parts of the number. The default is ".".
grouping-separator	Character used to separate groups of digits. The default is ",".
infinity	Text string used to represent infinite values. The default is "infinity".
minus-sign	Character used to represent negative values. The default is "-".
NaN	Text used to represent entries which are not numbers. The default is "NaN".
percent	Character used to represent numbers as percentages. The default is "%".
per-mille	Character used to represent numbers in parts per 1000. The default is "0/00".
zero-digit	Character used to indicate a required digit in the number format pattern. The default is "0".
digit	Character used to indicate an optional digit in the number format pattern. The default is "#".
pattern-separator	Character used to separate positive number patterns from negative number patterns in the number format. The default is ";".

Inserting Attribute Values

- XSLT expression inserted into HTML attribute value
- Syntax: <*tag attribute=”{XSLT expression}”* >
- Example: *<td rowspan=”{count(../../Items/Item)}”>*
- In this example, you can count the number of items in an XML node, then use it to format a table row dynamically in HTML

Working With Text Nodes And White Space

- White space:
  - Space devoid of any printable character
  - Space, tab, new line, carriage return
- Adjacent <xsl:value-of> elements will have results combined to eliminate white space
- <xsl:text> can be used to create white space:
  - Syntax: *<xsl:text>Text</xsl:text>*
  - Can only contain literal text

White Space & Controlling White Space

- Space -
- Tab - 	
- New line -

- Carriage return -
- Stripping space:
  - Remove text nodes from the result document that contain *only* white space
  - Syntax: *<xsl:strip-space elements="pattern">*
  - Use * as pattern to match all nodes

Controlling White space

- Preserving space:
  - Make sure that text nodes that contain only white space are not deleted
  - Syntax: *<xsl:preserve-space elements="pattern">*
  - Use * as pattern to match all nodes
- Normalize space:
  - Remove leading and trailing spaces
  - Syntax: *normalize-space(text)*

Using Variables

- User-defined name that stores a particular value or object
- Types:
  - number
  - text string
  - node set
  - boolean
  - result tree fragment

Using Variables

- Syntax: *<xsl:variable name="name" select="value"/>*
- Example: *<xsl:variable name="Months" select="12" />*
- Names are case-sensitive
- Value only set once upon declaration
- Unlike normal variables, cannot be re-defined after creation
- Enclose text strings in single-quotes

Using Variables

- Value can be XPath expression
- Boolean type:
  - Set value to expression that is true or false
- Result tree fragment type:
  - Syntax:

```
<xsl:variable name="Logo">

    <img src="logo.gif" width="300" height="100" />

</xsl:variable>
```

Referencing Variables

- Syntax: *$variable-name*
- Example: *$Months*
- Referencing tree fragments:
  - Do not use $variable-name
  - Use <xsl:copy> or <xsl:copy-of> to reference value

Copying

- <xsl:copy>
  - Syntax: *<xsl:copy use-attribute-sets="list" />*
  - Shallow copy: only node itself is copied
- <xsl:copy-of>
  - Syntax: *<xsl:copy-of select="expression"/>*
  - Deep copy: node and descendants are copied

Variable Scope

- Global:
  - Can be referenced from anywhere within the style sheet
  - Must be declared at the top level of the style sheet, as a direct child of the <xsl:stylesheet> element
  - Must have a unique variable name
- Local:
  - Referenced only within template
  - Can share name with other local or global variable

Using Parameters

- Similar to variables, but:
  - Value can be changed after it is declared
  - Can be set outside of scope
  - Can be sent into named templates
- Syntax: *<xsl:param name="name" select="value"/>*
- Example: *<xsl:param name="Filter" select="'C103'" />*
- To reference: *$param-name*

Setting Parameter Values Externally

- Depends on XSLT processor
- Some work by appending parameter value to url, though this usually requires JavaScript
- Command line processors allow external parameter setting:
  - MSXML
  - Saxon

Template Parameters

- Local in scope
- Created inside <xsl:template> element
- To pass parameter to template
  - place <xsl:with-param> element in <xsl:apply-templates> element
  - Syntax: *<xsl:with-param name="name" select="value"/>*
  - No error if calling param name does not match template param name

Introducing Functional Programming

- Functional programming language:
  - Relies on the evaluation of functions and expressions, not sequential execution of commands
  - Different from most other languages
  - There is usually always a ‘way’ to get something working with data passing in XML and XSLT, but requires tinkering and experimentation and may not always be cross browser compatible

Functional Programming Principles

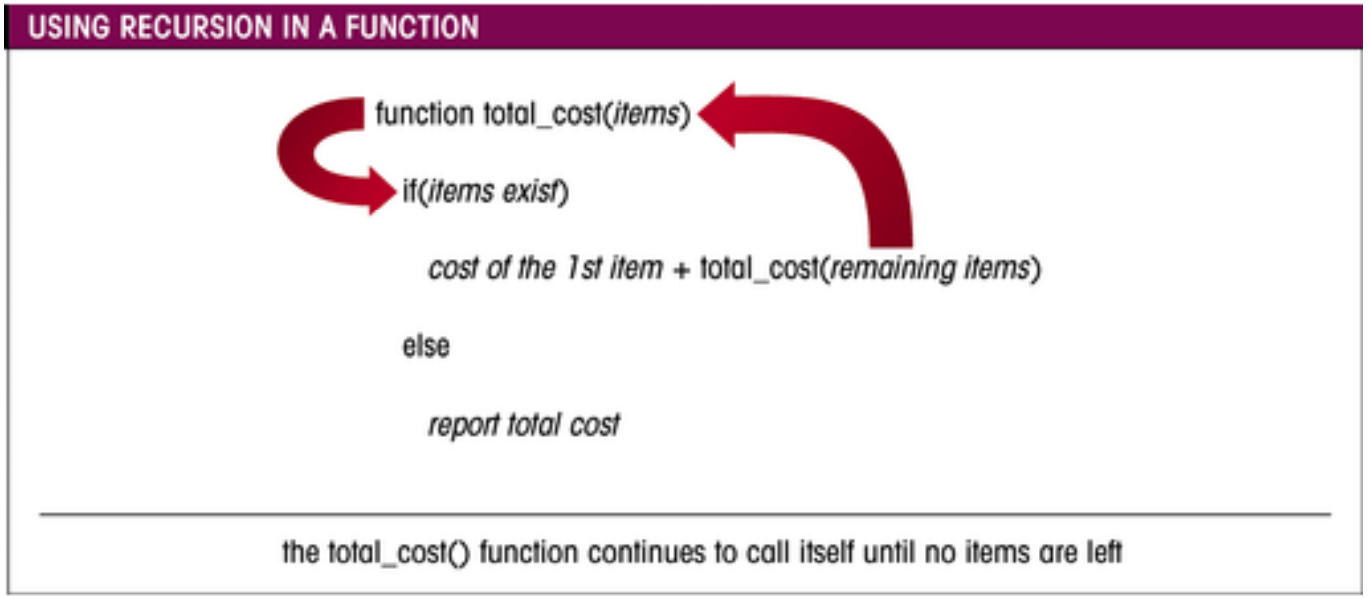
- Main program consists entirely of functions with well-defined inputs
- Results of program are defined in terms of function outputs
- No assignment statements; when a variable is declared, its value cannot be changed
- Order of the execution is irrelevant

Functional Programming

- Easier to maintain and less susceptible to error
- No order of execution
- Each time a function is called, it does the same thing, regardless of how many times it has already been called, or the condition of other variables in the program
- Important to think about the desired end result, rather than the sequential steps needed to achieve that effect

Recursion

- Takes the place of looping structures in functional programming
- Function calls itself



Using Named Templates

- Template not associated with node set
- Collection of functions and commands that are accessed from other templates in the style sheet
- Syntax:

```
<xsl:template name="name">
        XSLT elements
</xsl:template>
```

Calling Named Templates

- Syntax:

```
<xsl:call-template name="name">
    <xsl:with-param />
    <xsl:with-param />
    ...
</xsl:call-template>
```

Writing A Recursive Template

- Templates that call themselves, usually passing along a new parameter value with each call
- Needs to have a stopping condition
  - Expressed in an if statement or a choose statement
  - If missing, will call itself without end (runaway loop)

Writing A Recursive Template

- Syntax with <xsl:if>:

```
<xsl:template name="template_name">
  <xsl:param name="param_name" select="default_value" />
  ...
  <xsl:if test="stopping_condition">
  ...
    <xsl:call-template name="template_name">
    <xsl:with-param name="param_name" select="new_value" />
    </xsl:call-template>
  ...
  </xsl:if>
</xsl:template>
```

Writing A Recursive Template

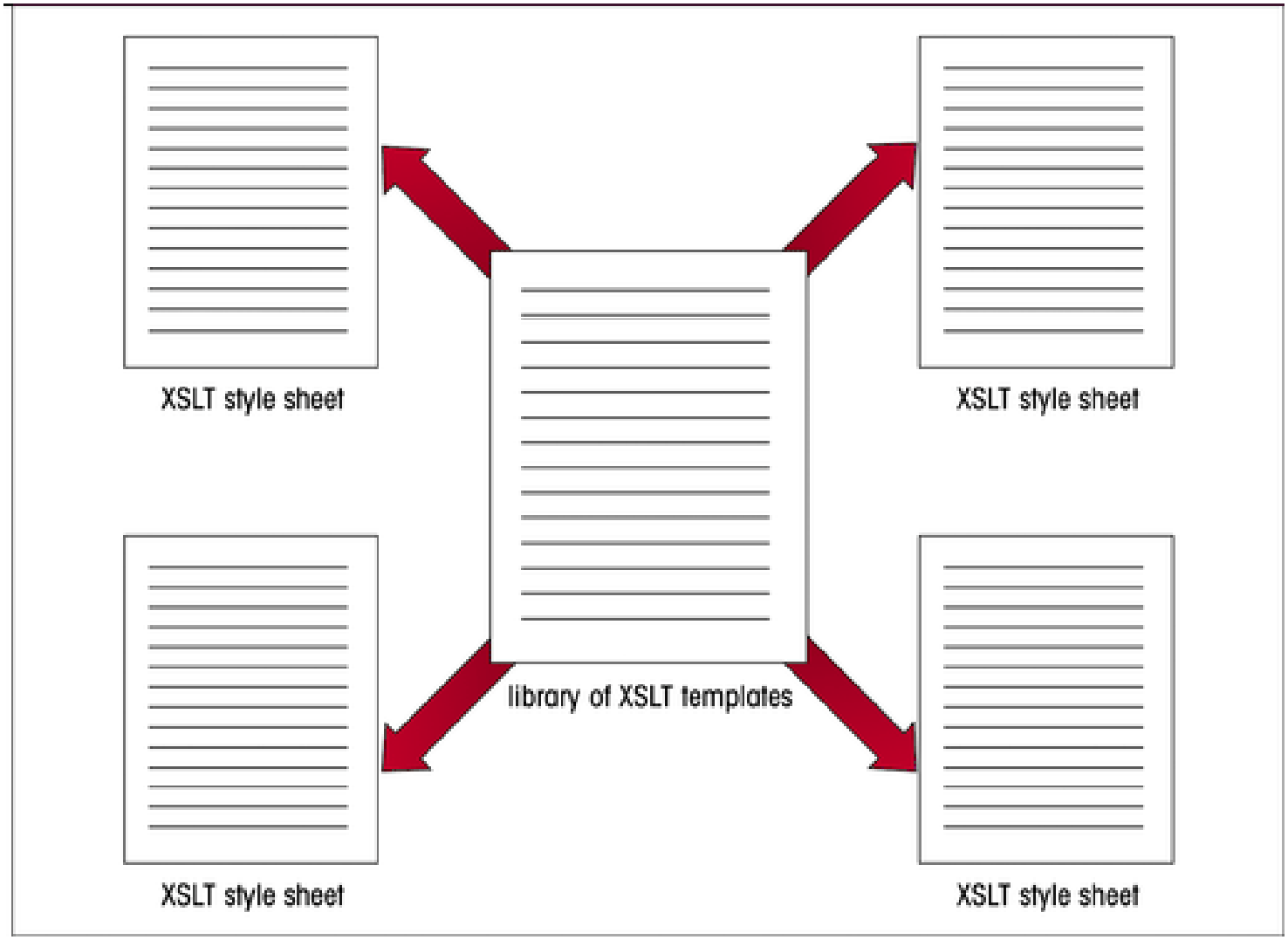
- Syntax with <xsl:choose>

```
<xsl:template name="template_name">
  <xsl:param name="param_name" select="default_value" />
  ...
  <xsl:choose>
  <xsl:when test="stopping_condition">
  ...
  </xsl:when>
  <xsl:otherwise>
  ...
    <xsl:call-template name="template_name">
    <xsl:with-param name="param_name" select="new_value" />
    </xsl:call-template>
  ...
  </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Recursive Template

```
<xsl:template name="total_cost">
   <xsl:param name="list" />
   <xsl:param name="total" select="0" />
   <xsl:choose>
      <xsl:when test="$list">
         <xsl:variable name="first" select="$list[1]" />
         <xsl:call-template name="total_cost">
            <xsl:with-param name="list" select="$list[position() > 1]" />
            <xsl:with-param name="total"
                 select="$first/@Qty * $first/@Price + $total" />
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="format-number($total, '$#,#00.00')" />
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Working with Multiple Style Sheets



Working with Multiple Style Sheets

- Use to create a library of XSLT code
- <xsl:include>
  - Syntax: *<xsl:include href="URL" />*
  - Same as inserting the components of included sheet directly into including file
  - Does not perform a direct text copy
  - Performs logical copy
  - If naming conflict occurs, last occurrence of template is used

Working with Multiple Style Sheets

- <xsl:import>
  - Syntax: *<xsl:import href="URL" />*
  - Must be at the top level of the style sheet; must be first child of <xsl:stylesheet>
  - If name conflicts occur, importing sheet takes precedence

Working with Extension Functions

- Provide extended functionality specific to XSLT processor
- Extension functions - extend the list of functions available to XPath and XSLT expressions
- Extension elements - extend the list of elements that can be used in an XSLT style sheet
- Extension attributes - extend the list of attributes associated with XSLT elements
- Extension attribute values - extend the data types associated with XSLT attributes

Working with Extension Functions

- Check processor documentation for support
- EXSLT:
  - Proposed common set of extensions
  - Not fully supported by all processors
  - Some functions supported by:
    - 4XSLT
    - saxon
    - jd.xslt
    - libxslt
    - Xalan-J

Working with Extension Elements and Attributes

- Must be associated with a namespace
- Saxon XSLT processor:
  - Allows variables to be changed:
    - *<xsl:variable name="name" saxon:assignable="yes" />*
    - *<saxon:assign name="name" select="expression" />*
  - Allows program loops:

<saxon:while test="condition">

XSLT and literal result elements

</saxon:while>

Summary

- <xsl:number> and position() are used to number elements
- XPath provides mathematical and string operators
- Standard mathematical operators (+, -, etc.) can be used in XSLT
- Number formatting uses format-number() function
- XSLT expressions are placed in HTML attributes using {} operators

Summary

- White space characters are controlled with various XSLT elements and XPath functions
- Variables and parameters are used to supply user-defined values
- XSLT is a functional programming language
- Recursive templates provide looping logic in XSLT
- EXSLT provides a set of extension functions, elements, attributes and attribute values