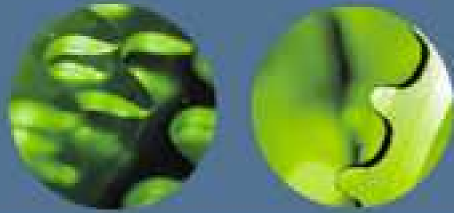
Exploring the New Features of XSLT 2.0

Priscilla Walmsley
Managing Director, Datypic
http://www.datypic.com
pwalmsley@datypic.com

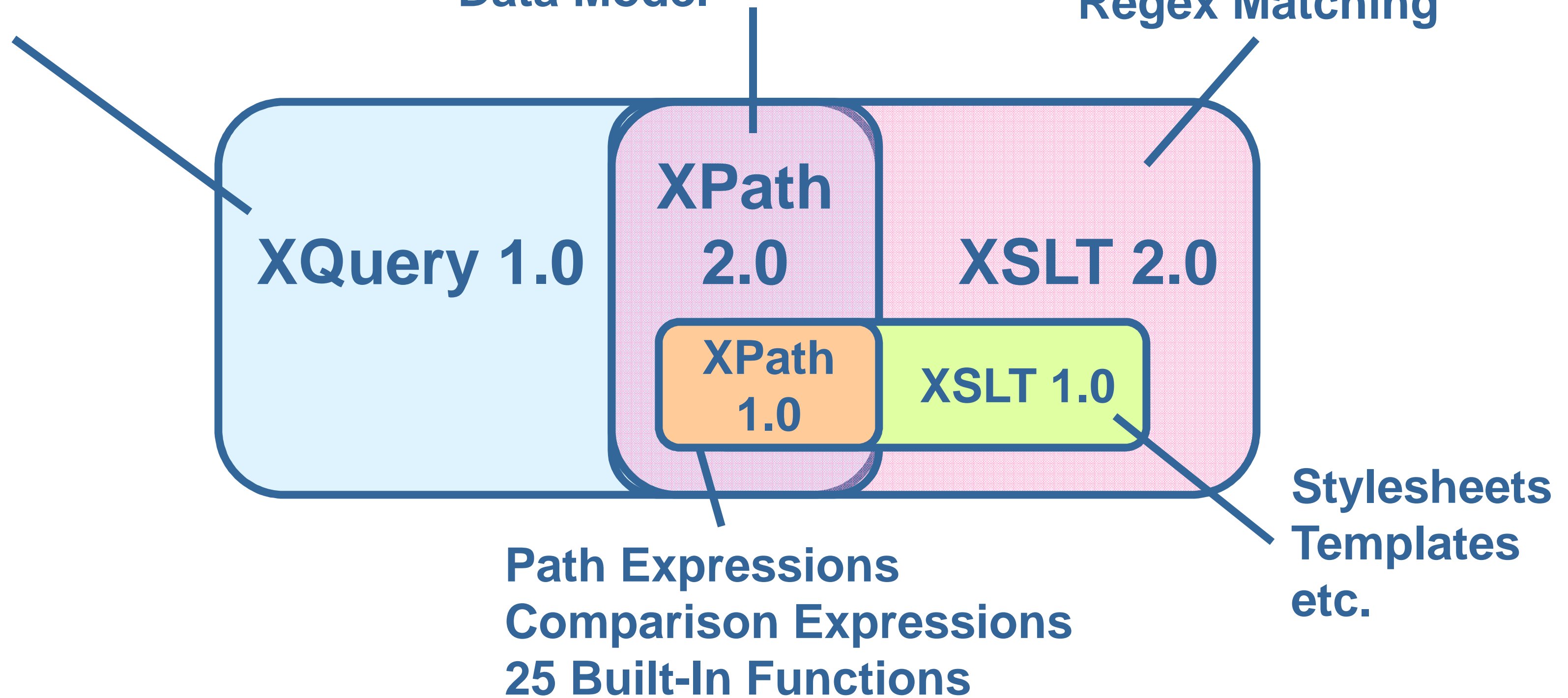


W3C Standards for Querying/Transformation

FLWOR Expressions
XML Constructors
Query Prolog
User-Defined Functions

Conditional Expressions
Arithmetic Expressions
Quantified Expressions
100 Built-In Functions
Data Model

Grouping
User-Defined Functions
Regex Matching

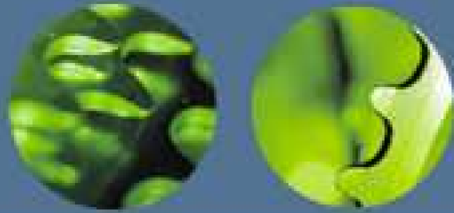


Stylesheets
Templates
etc.

Path Expressions
Comparison Expressions
25 Built-In Functions

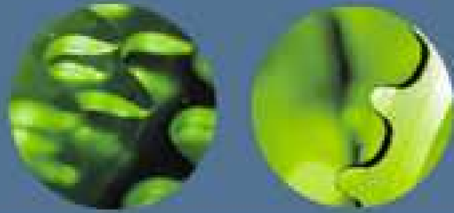
© 2008 Datypic http://www.datypic.com

XPath Differences



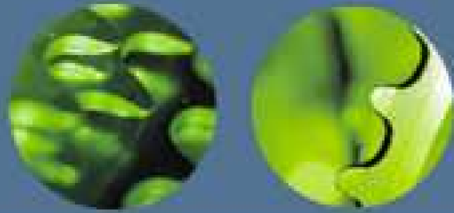
Data Model Differences

- Nodes
  - root nodes are now "document nodes"
  - namespace nodes are deprecated
- Atomic values
  - more types, e.g. `xs:integer, xs:date`
- Sequences (formerly node-sets)
  - are ordered
  - can contain duplicates
  - can contain atomic values as well as nodes



Path Expressions in XPath 2.0

- Basic syntax is still the same
  - Same steps, predicates, node tests
  - Same axes
    - except that the namespace axis is deprecated
- Key new features
  - Expressions as steps
  - Paths that return atomic values
  - New operators that can be used in predicates



Expressions as Steps

- Expressions can be used as steps
  - not just node tests

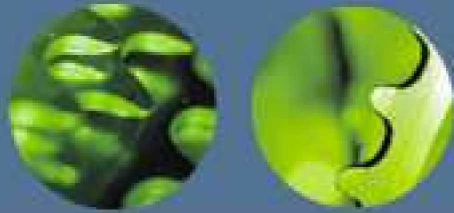
```
product/(number | name)
```

```
product/(if (desc) then desc else name)
```

```
$catDoc/catalog/product
```

```
$prods[position() > 3]
```

```
//product/dty:ordersForProd(.)/orderID
```



Paths Returning Atomic Values

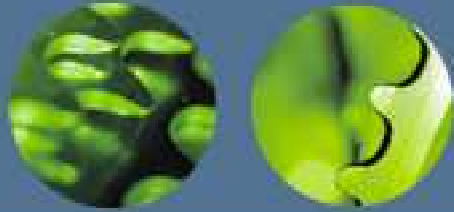
- The last step in a path can now return an atomic value

```
product/name/substring(.,1,9)
```

```
sum(//item/(@price * @qty))
```

- But only the last step

```
product/name/substring(.,1,9)/replace(.,'x','y')
```



Comparing Values of Different Types

- In XPath 2.0, the comparison operators do not just apply to numbers.
- All of the following return `true`:

```
99 < 100
```

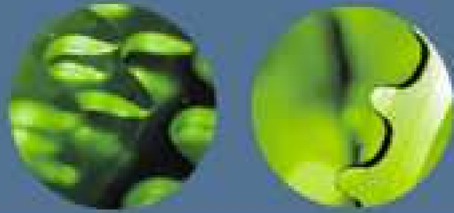
```
'100' < '99'
```

Caution! Backward incompatibility

```
'a' < 'aaa'
```

```
'aaa' < 'bbb'
```

```
current-date() < xs:date("2009-06-01")
```



Arithmetic Expressions: What's New?

- Arithmetic can be performed on dates and durations as well as numbers

```
current-date() + xs:duration("P1M")
```

- Missing value returns empty sequence rather than NaN

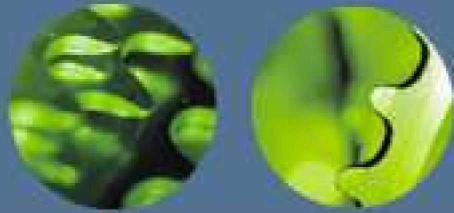
```
product[1]/foo * 2
```

← returns ()

- New `idiv` operator (integer division)

```
14 idiv 4
```

← returns 3



New Operators for Combining Sequences

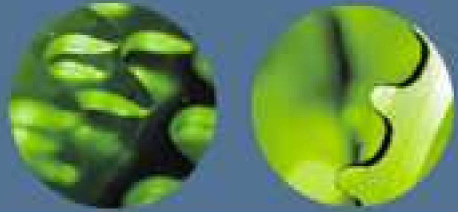
- concatenation
  - duplicates are not removed, order is retained

```
($seq1, $seq2)
```

- **`union`**, **`intersect`** and **`except`**
  - duplicates are removed
  - results are sorted in document order
  - work on sequences of *nodes* only, not atomic values

```
$seq1 union $seq2
$seq1 | $seq2
$seq1 intersect $seq2
$seq1 except $seq2
```

© 2004 - 2008 Datypic http://www.datypic.com

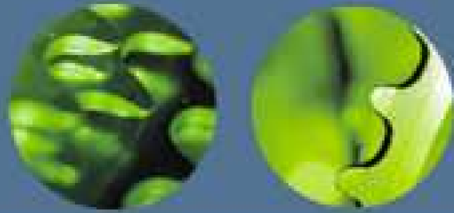


Range Expressions

- Create sequences of consecutive integers
- Use `to` keyword
  - `(1 to 5)` evaluates to a sequence of 1, 2, 3, 4 and 5
- Useful to iterate a specific number of times

```
<xsl:for-each select="1 to 5"> ...
```

```
<xsl:for-each select="1 to $count"> ...
```



Node Comparison Operators

- To compare nodes based on document order, use
  - << for precedes
  - >> for follows

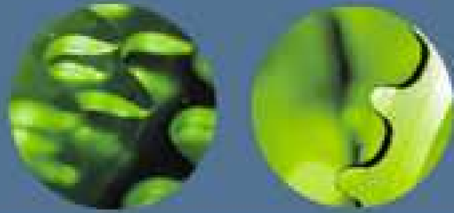
```
h[1] >> p[1]
```

- `is` operator to see if two nodes are the same node

```
$thisEl is p[1]
```

returns false

- For nodes only
  - no document order on atomic values

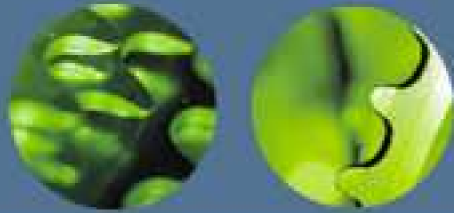


Conditional Expressions

- if-then-else syntax

```
<xsl:variable name="heading"
select="if (exists(desc))
          then desc
          else name"/>
```

- else is always required
  - but it can be just `else ()`
- More compact alternative to `xsl:choose`



For Expressions

- Simplified version of XQuery FLWOR expressions
  - only one for clause, no let or where

```
<xsl:variable name="prodNames"
select="for $aName in //product/substring(name,1,9)
        return replace($aName,'x','y')"/>
```

- More compact alternative to **`xsl:for-each`**

© 2008 Datypic http://www.datypic.com

New Built-In Functions: String-Related

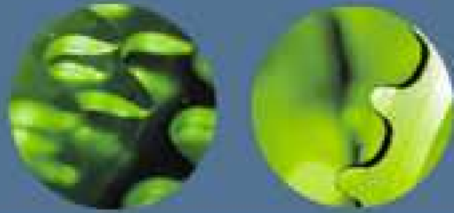
- Regular expression matching
  - `matches, replace, tokenize`
- **`ends-with`**

```
ends-with("XPath","th") ==> true
```

- **`lower-case, upper-case`**

```
upper-case("XPath") ==> XPATH
```

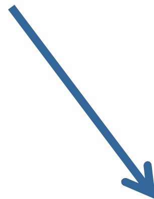
- Escaping and normalizing
  - `normalize-unicode, iri-to-uri, escape-html-uri`



`string-join` function

- Use the **`string-join`** function instead of looping through strings

```
<xsl:for-each select="//name">
  <xsl:value-of select="."/>
  <xsl:if test="position() != last()">, </xsl:if>
</xsl:for-each>
```



```
<xsl:value-of select="string-join(//name,', ')"/>
```

New Built-In Functions: Date-Related

- current date/time
  - `current-date`, `current-time`, `current-dateTime`

```
current-date() ==> 2008-12-09
```

- getting date components
  - e.g. `month-from-date`, `seconds-from-time`

```
month-from-date(xs:date('2008-12-09')) ==> 12
```

- `format-date`

```
format-date(xs:date('2008-12-09'),
    '[D1o] [MNn], [Y]', 'en', (), ())
                                 ==> 9th December, 2008
```

New Built-In Functions: Other

- average (**avg**)

```
avg($prods/price) ==> 12.54
```

- minimum and maximum (**min**, **max**)

```
max($prods//price) ==> 499.99
```

```
min($people//firstname/string(.)) ==> "Aaron"
```

not just for numbers

- **distinct-values**

```
distinct-values($prods//@dept)
    ==> ("WMN","ACC","MEN")
```

The `deep-equal` Function

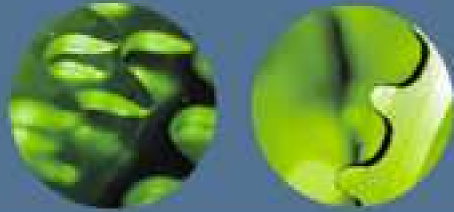
- **`deep-equal`** tests whether two nodes have the same contents

```
<product dept='MEN' id='P123'>
  <number>784</number>
</product>
<product id='P123' dept='MEN'><!--comment-->
  <number>784</number>
</product>
```

```
deep-equal( product[1], product[2]) ==> true
```

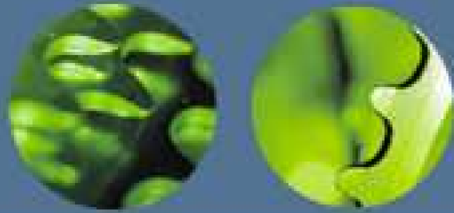
XSLT 2.0

Grouping



Grouping

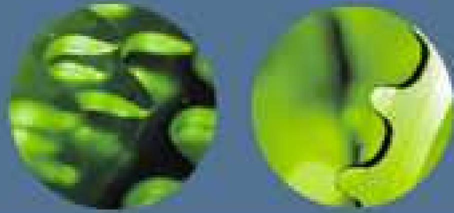
- Difficult in XSLT 1.0; usually used keys
- `xsl:for-each-group` element allows you to iterate through groups
  - `select` attribute identifies items to group
  - `group-by` attribute specifies the grouping key
- Two functions can be used within `for-each-group`:
  - `current-group()` returns members of current group
  - `current-grouping-key()` returns the current grouping key



Grouping by Value

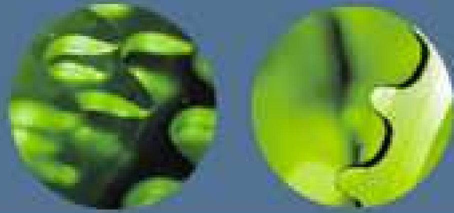
```
<xsl:template match="/" >
 <RESULTS>
  <xsl:for-each-group select="catalog/product"
                      group-by="@dept" >
   <xsl:sort select="current-grouping-key()" />
   <DEPT name="{current-grouping-key()}"
         prodCount="{count(current-group())}" >
   </DEPT>
  </xsl:for-each-group>
 </RESULTS>
</xsl:template>
```

```
<RESULTS>
 <DEPT name="ACC" prodCount="2"/>
 <DEPT name="MEN" prodCount="1"/>
 <DEPT name="WMN" prodCount="1"/>
</RESULTS>
```



Grouping by Sequence

- Instead of **`group-by`**, you can use:
  - **`group-adjacent`**
    - groups adjacent items with the same key together
  - **`group-starting-with`**
    - creates a group of items starting with the specified element
  - **`group-ending-with`**
    - creates a group of items ending with the specified element



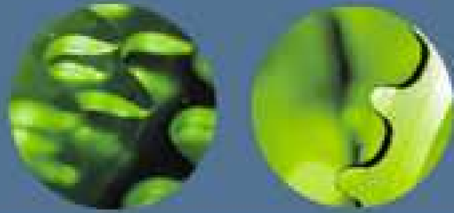
Grouping by Sequence

```
<body>
 <h1>Chapter 1</h1>
 <p>This chapter...</p>
 <h2>Section 1.1</h2>
 <p>In this section...</p>
 <p>More text</p>
 <h2>Section 1.2</h2>
 <p>In this section...</p>
</body>
```

Input document

```
<section level="1">
  <section level="2">
   <heading>Chapter 1</heading>
   <p>This chapter...</p>
  </section>
  <section level="2">
   <heading>Section 1.1</heading>
   <p>In this section...</p>
   <p>More text</p>
  </section>
  <section level="2">
   <heading>Section 1.2</heading>
   <p>In this section...</p>
  </section>
</section>
```

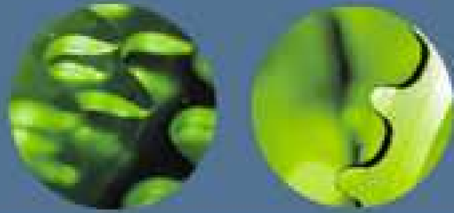
Output document



Grouping by Sequence

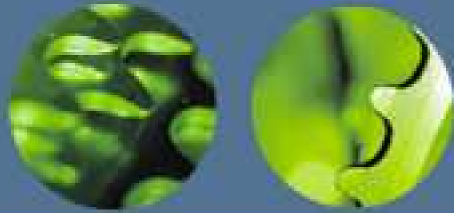
```
<xsl:template match="/" >
  <xsl:for-each-group select="body/*" group-starting-with="h1" >
   <section level="1" >
    <xsl:for-each-group select="current-group()"
                        group-starting-with="h2" >
     <section level="2" >
      <xsl:apply-templates select="current-group()" />
     </section>
    </xsl:for-each-group>
   </section>
  </xsl:for-each-group>
</xsl:template>
```

User-Defined Functions



User-Defined Functions

- Define your own functions using **`xsl:function`**
  - **`xsl:param`** child is used to define a parameter
- Similar to named templates but more useful
- Contents of **`xsl:function`** are the results returned
  - might be an **`xsl:copy-of`**, an **`xsl:value-of`**, a constructed element, etc.
- Benefits
  - Reuse
  - Clarity
  - Recursion



Function Declaration Example

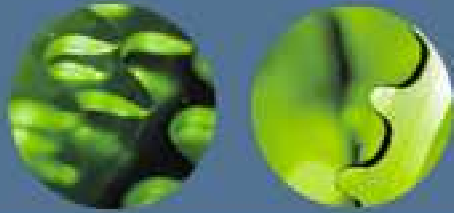
```
<xsl:function name="my:substring-after-last"
              as="xs:string">
  <xsl:param name="string" as="xs:string"/>
  <xsl:param name="delim" as="xs:string"/>
  <xsl:value-of select="
    if (contains ($string, $delim))
    then my:substring-after-last(
       substring-after($string, $delim), $delim)
    else $string"/>
</xsl:function>
```

return type

parameters

```
<xsl:value-of select="my:substring-after-last(
  'abc,def,ghi', ',')"/>    ==> 'ghi'
```

function call

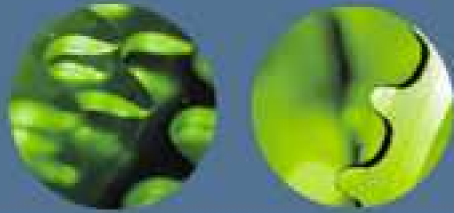


Convenient Uses for Functions

- Unlike named templates, functions can be very useful in places where only a simple XPath expression or pattern is allowed, e.g.:
  - The **`select`** or **`group-by`** attributes of **`xsl:for-each-group`**
  - The **`select`** attribute of **`xsl:sort`**
  - The **`match`** attribute of **`xsl:template`**

```
<xsl:template match="*[my:contains-word(.,'Section')]">
```

Regular Expression Capabilities



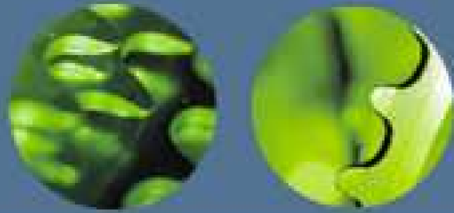
The `matches` Function

- **`matches`**
  - whether a string matches a regular expression

```
matches("query", "^qu")   ==> true
```

  - uses the XML Schema regex syntax (similar to Perl)
  - optional third "flags" argument allows for interpretation of regular expression
    - case sensitivity
    - multi-line mode
    - whitespace sensitivity

© 2008 Datypic http://www.datypic.com



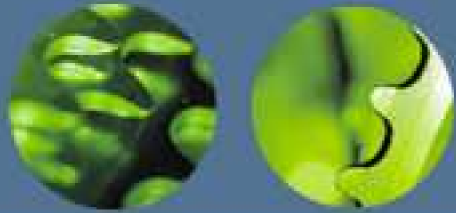
The `tokenize` Function

- `tokenize`
  - delimiter specified as a regular expression
  - returns a sequence of strings

```
tokenize("a b c", "\s")
   ==> ("a", "b", "c")
```

```
tokenize("2006-12-25T12:15:00", "[\-T:]")
   ==> ("2006","12","25","12","15","00")
```

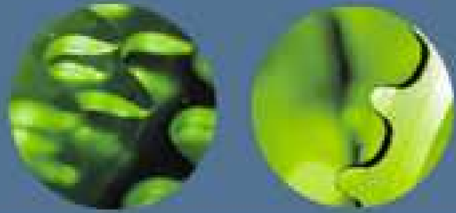
© 2008 Datypic http://www.datypic.com



The `replace` Function

- Arguments are:
  - the string to be manipulated
  - the regular expression
  - the replacement string

`replace("query", "r", "as")`	`queasy`
`replace("query", "qu", "quack")`	`quackery`
`replace("query", "[ry]", "l")`	`quell`
`replace("query", "[ry]+", "l")`	`quel`

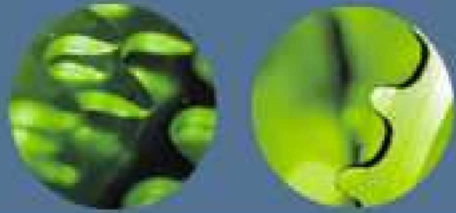


Replacing with Subexpressions

- Use `$1, $2,` etc. in replacement string to insert strings that matched parenthesized subexpressions

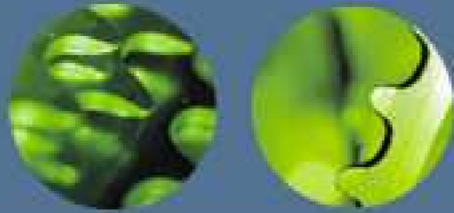
```
replace("Chap 2...Chap 3...Chap 4...",
        "Chap (\d)", "Sec $1.0")
   ==> "Sec 2.0...Sec 3.0...Sec 4.0..."
```

```
replace("2006-10-18", "\d{2}(\d{2})-(\d{2})-
(\d{2})", "$2/$3/$1")
   ==> "10/18/06"
```



String Handling Using `xsl:analyze-string`

- **`xsl:analyze-string`** element splits string into matching and non-matching parts, based on a regex
  - **`select`** attribute specifies the string
  - **`regex`** attribute specifies regular expression
  - **`xsl:matching-substring`** child specifies what to do with matching parts
  - **`xsl:non-matching-substring`** child specifies what to do with non-matching parts
- More powerful than **`matches`** or **`replace`**

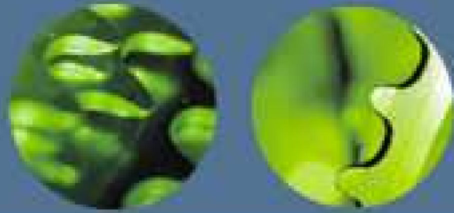


xsl:analyze-string Example

```
<xsl:function name="my:markUpPhone">
 <xsl:param name="theText"/>
 <xsl:analyze-string select="$theText"
                     regex="[0-9]{{3}}/[0-9]{{3}}-[0-9]{{4}}">
  <xsl:matching-substring>
   <xsl:element name="phone">
    <xsl:value-of select="."/>
   </xsl:element>
  </xsl:matching-substring>
  <xsl:non-matching-substring>
   <xsl:copy/>
  </xsl:non-matching-substring>
 </xsl:analyze-string>
</xsl:function>
```

```
can be reached at 231/555-1212 or...
```

```
can be reached at <phone>231/555-1212</phone> or...
```



The regex-group Function

```
<xsl:function name="my:markUpPhone">
 <xsl:param name="theText"/>
 <xsl:analyze-string select="$theText"
                     regex="([0-9]{{3}})/([0-9]{{3}}-[0-9]{{4}})">
  <xsl:matching-substring>
   <xsl:element name="phone">
    <areaCode><xsl:value-of select="regex-group(1)"/></areaCode>
    <number><xsl:value-of select="regex-group(2)"/></number>
   </xsl:element>
  </xsl:matching-substring>  ....
```

```
can be reached at 231/555-1212 or...
```

```
can be reached at
<phone><areaCode>231</areaCode><number>555-
1212</number></phone> or...
```

© 2008 Datypic http://www.datypic.com

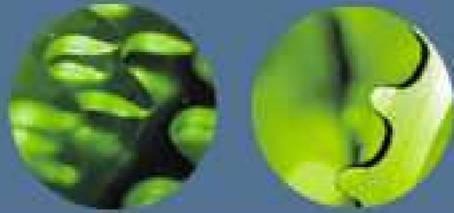
Inputs and Outputs

The `collection` function

- The `collection` function references a collection via a URI
- Returns a sequence of document nodes

```
collection("mycoll.xml")
```

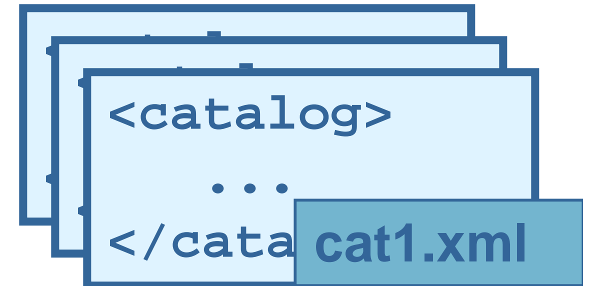
- Collections are implementation defined
  - for example:
    - Saxon accepts the URI of an XML document that lists the documents that make up the collection



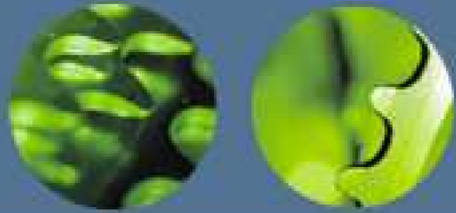
Saxon Collection Example

cats.xml

```
<collection>
   <doc href="./catalogs/cat1.xml"/>
   <doc href="./catalogs/cat2.xml"/>
   <doc href="./catalogs/cat3.xml"/>
   <doc href="./catalogs/cat4.xml"/>
   <doc href="./catalogs/cat5.xml"/>
</collection>
```

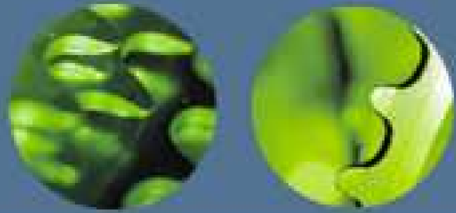



```
<xsl:for-each select="collection('cats.xml')">
  <xsl:apply-templates select="catalog"/>
</xsl:for-each>
```



Multiple Result Documents

- `xsl:result-document` allows you to create a new result document
  - `href` attribute indicates URI
- Useful for splitting documents
  - multiple HTML pages e.g. by chapter or number of records
  - multiple separate transactions in a pipeline



Multiple Result Documents Example

```
<xsl:for-each select="catalog/product">
 <xsl:result-document href="prod_{number}.xml">
   <xsl:copy-of select="." />
 </xsl:result-document>
</xsl:for-each>
```

prod_557.xml

```
<product dept="ACC">
    ...
</product>
```

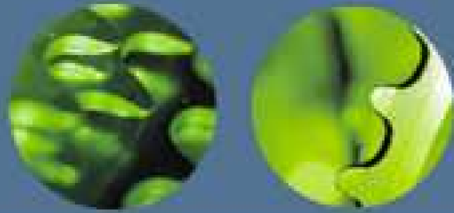
prod_563.xml

```
<product dept="MEN">
    ...
</product>
```

prod_443.xml

```
<product dept="WMN">
    ...
</product>
```

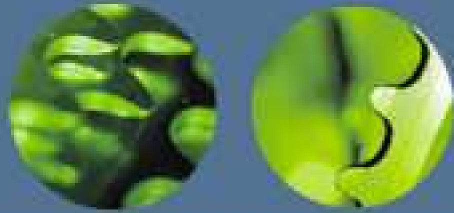
© 2008 Datypic http://www.datypic.com



The `unparsed-text` Function

- **`unparsed-text`** function allows you to open any text document as a string
- When combined with **`xsl:analyze-string`**, can allow XSLT to be used to convert non-XML formats to XML

© 2008 Datypic http://www.datypic.com



unparsed-text Example

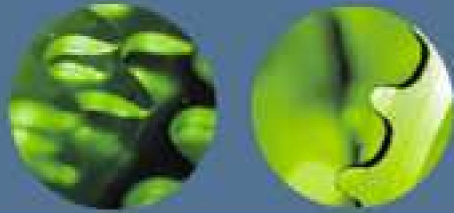
```
<xsl:template match="/">
  <settings>
    <xsl:analyze-string regex="(.*)=(.*)\r\n"
         select="unparsed-text('../build.properties')">
      <xsl:matching-substring>
         <xsl:element name="{regex-group(1)}">
           <xsl:value-of select="regex-group(2)"/>
         </xsl:element>
      </xsl:matching-substring>
    </xsl:analyze-string>
  </settings>
</xsl:template>
```

```
#comment
urlDir=xq
prefix=functx
suffix=xq
```

→

```
<settings>
   <urlDir>xq</urlDir>
   <prefix>functx</prefix>
   <suffix>xq</suffix>
</settings>
```

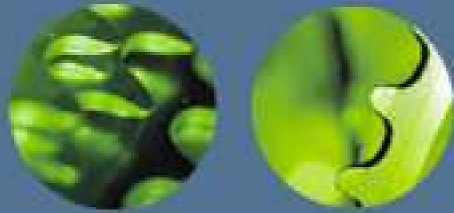
Types and Schemas



The XSLT/XPath 2.0 Type System

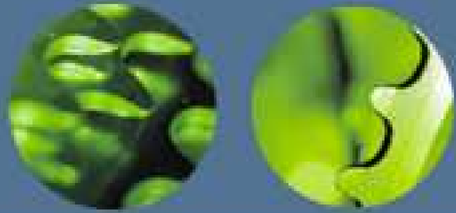
- 2.0 is more strongly typed than 1.0
- The type system is based on XML Schema
  - built-in types from XML Schema (e.g. `xs:integer, xs:string, xs:date`) are automatically built into the language
  - other types can be imported from a schema
- Values of a certain type can be constructed
  - e.g. `xs:date("2004-12-15")`

© 2008 Datypic http://www.datypic.com



Strong Typing: Pros and Cons

- Pro
  - easy identification of static errors
    - saves time debugging and testing
    - identifies errors that even testing may not ever uncover
  - may allow for better optimization
- Con
  - adds complexity because explicit casting is required in some cases



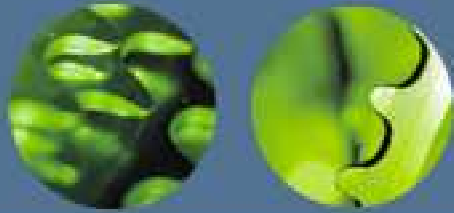
Do I Have to Pay Attention to Types?

- Usually, no, not if you don't want to.
  - without a schema, your XML data is "untyped"
  - in most expressions, untyped values are cast to the expected types

```
product/price + 23
```

```
sum(product/price)
```

if `price` is untyped, it is cast to a numeric type automatically



When You Have to Pay Attention to Types

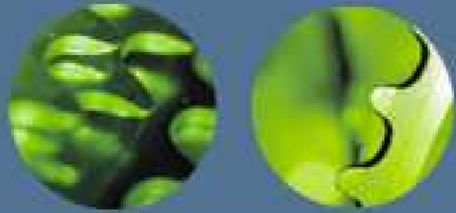
- use **number** and **string** functions (or type constructors) to cast between types

```
substring(string(current-date()),1,4)
```

without `string` function, it's a type error because the `substring` function is expecting `xs:string` values only; `current-date()` returns an `xs:date`.

```
<xsl:for-each select="//product">
 <xsl:if test="number(price) < number(discount)">
...
```

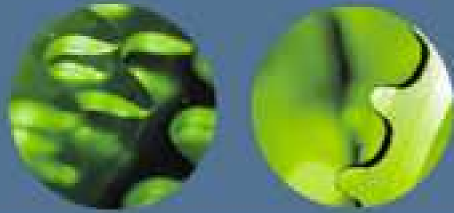
without `number` function, 53 will be greater than 144



Declaring Types in XSLT

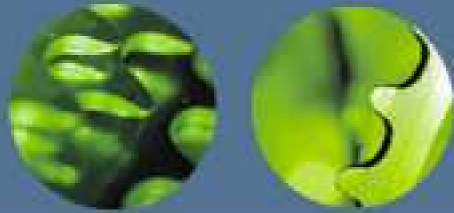
- You can specify types in your XSLT using an **as** attribute
  - Values must conform to that type
  - Helpful for debugging

```
<xsl:variable name="prods" as="node()*" select="product"/>

<xsl:param name="prodCount" as="xs:integer?"/>

<xsl:function name="getPrice" as="xs:decimal">...

<xsl:template name="createList" as="element(ul)">...
```



Schemas and XSLT 2.0

- Schemas can pass type information to the stylesheet
- Use of schemas is *optional*
- You can:
  - validate input and/or result documents
  - import schema documents, which lets you:
    - know when your XSLT violates the schema
    - do special processing based on types



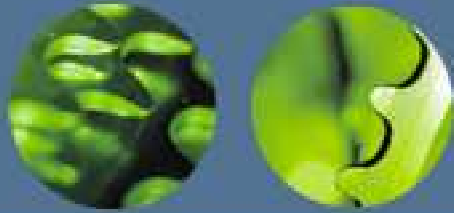
Using Schemas to Catch Static Errors

```
<xsl:stylesheet .....
<xsl:import-schema
  namespace="http://www.datypic.com/prod"
  schema-location="prod.xsd" />

<xsl:template match="schema-element(catalog)">
  <xsl:for-each select="produt">
    <xsl:sort select="product/number"/>
    <xsl:value-of select="name + 1"/>
  </xsl:for-each>
</xsl:template>
```

misspelling

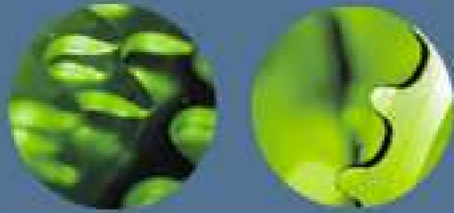
invalid path; product will never have product child

type error: name is declared to be of type xs:string, so cannot be used in an add operation



Special Processing Based on Type

```
<xsl:stylesheet .....
<xsl:import-schema
  namespace="http://www.datypic.com/prod"
  schema-location="prod.xsd" />

<xsl:template match="element(*,USAddressType)">
   .... <xsl:value-of select="city"/>
        <xsl:value-of select="zipCode"/>
</xsl:template>

<xsl:template match="element(*,UKAddressType)">
   .... <xsl:value-of select="postCode"/>
        <xsl:value-of select="city"/>
</xsl:template>
```



Using Schemas to Validate Results

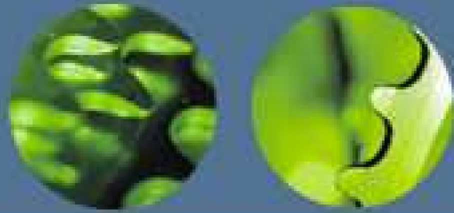
```
<xsl:stylesheet .....
<xsl:import-schema
  namespace="http://www.datypic.com/res"
  schema-location="res.xsd" />

<xsl:template match="catalog">
 <xsl:result-document validation="strict">
  <res:root>
    <xsl:apply-templates/>
  </res:root>
 </xsl:result-document>
</xsl:template>
```

target namespace

schema location

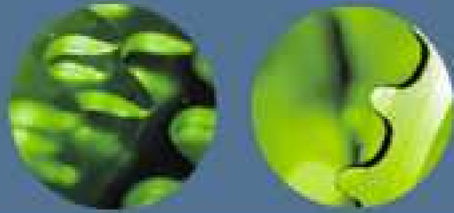
explicit validation

Miscellaneous Enhancements



Temporary Trees

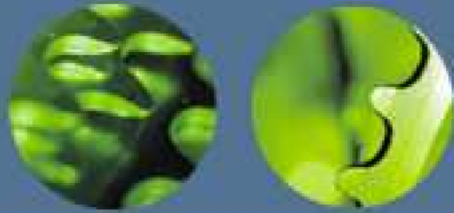
- Variables can contain element structures that can now be accessed using XPath
- Useful for:
  - defining lookup tables
  - simplifying queries, especially with multi-step processes
- In 1.0 this was generally handled by a `node-set` extension function



Temporary Tree Example

```
<xsl:variable name="dept_names" >
 <Dep code="ACC" name="Accessories" />
 <Dep code="MEN" name="Men's" />
 <Dep code="WMN" name="Women's" />
</xsl:variable>
<xsl:for-each-group select="catalog/product" group-
by="@dept" >
  <DEPT name="{$dept_names/Dep[@code=current-
grouping-key()]/@name}">...</DEPT>
 </xsl:for-each-group>
```

```
<DEPT name="Accessories">...</DEPT>
<DEPT name="Men's">...</DEPT>
<DEPT name="Women's">...</DEPT>
```



Tunnel Parameters

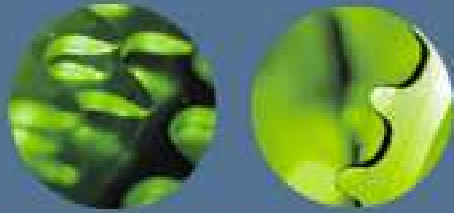
- Parameters are passed from template to template implicitly

```
<xsl:template match="doc">
  <xsl:apply-templates>
     <xsl:with-param name="docID" select="@id"
                     tunnel="yes"/>
  </xsl:apply-templates>
</xsl:template>
```

```
<xsl:template match="chap">
   <xsl:apply-templates/>...
```

no param

```
<xsl:template match="section">
   <xsl:param name="docID" tunnel="yes"/>
    <xsl:value-of select="$docID"/>...
```



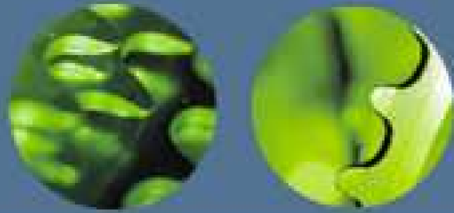
Modes in XSLT 2.0

- Multiple modes can be specified in template
- `#current` can be used to pass current mode

```
<xsl:template match="topic" mode="mainBody index">
  <!-- do something -->
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

- `#all` keyword can be used to match all modes

```
<xsl:template match="topic" mode="#all">
  <!-- do something -->
</xsl:template>
```



New `select` Attribute

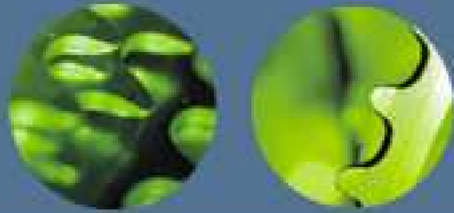
- A `select` attribute can appear on `xsl:attribute`, `xsl:message`, `xsl:processing-instruction`

```
<xsl:attribute name="class" select="'boldpara'"/>
```

```
<xsl:attribute name="class"
    select="if ($status='bold')
            then 'boldpara'
            else 'para'"/>
```

```
<xsl:message select="$theErrorNumber"/>
```

© 2008 Datypic http://www.datypic.com



New separator Attribute

- A **separator** attribute can appear on **xsl:attribute, xsl:value-of**
  - If not specified, separator is a space
  - XSLT 1.0 would just take the first value!

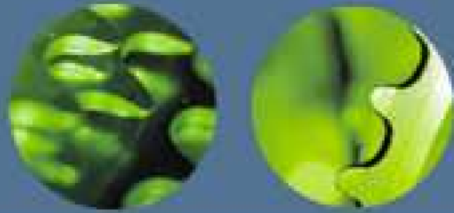
Caution! Backward incompatibility

```
<nums>
  <xsl:attribute name="values" select="//num"
  separator=","/>
</nums>
```

```
<nums values="123,144,344,456"/>
```

```
<nums>
  <xsl:value-of select="//num" separator=""/>
</nums>
```

```
<nums>123144344456</nums>
```



XSLT 2.0 Resources

- XSLT 2.0 Recommendation:
  - **`http://www.w3.org/TR/xslt20`**
- Book:
  - Kay, Michael. *XSLT 2.0 and XPath 2.0 Programmer's Reference.* Wrox, 2008.
- Reusable XSLT 2.0 functions:
  - **`http://www.xsltfunctions.com`**
- Saxon:
  - **`http://www.saxonica.com`**

Thank you for your interest.

For more information please contact me at:

Email: pwalmsley@datypic.com
Website: http://www.datypic.com