5.1 XML-QL

- combines XML syntax with query language techniques from Chapter 4

- Uses path expressions and patterns to extract data from input XML data

- It has templates which show how the output XML data of the query will be constructed

- where-construct syntax instead of the more familiar select-from-where
Example:

where

<book>
   <publisher><name><Morgan Kaufmann</name></publisher>
   <title> $T </title>
   <author> $A </author>
</book>  in "www.a.b.c/bib.xml

construct $A;

<book>...</book> is the pattern
$T and $A are variables
The query processor will match the pattern in all
possible ways with the data and bind the variables $T and $A;

For each binding, the query processor will construct $A
Constructing new XML data:

where
<book>
  <publisher><name><Morgan Kaufmann</name></publisher>
  <title> $T $ </title>
  <author> $A $ </author>
</book> in "www.a.b.c/bib.xml"

construct
<result>
  <author> $A $ </author>
  <title> $T $ </title>
</result>
Consider the following XML data:

```xml
<book year="1991">
    <!-- A good introductory text -->
    <title> An Introduction to Parallel Algorithms and Architectures </title>
    <author><lastname>Leighton</lastname></author>
    <publisher><name>Morgan Kaufmann</name></publisher>
</book>

<book year="1995">
    <title> Active Database Systems </title>
    <author><lastname>Ceri</lastname></author>
    <author><lastname>Widom</lastname></author>
    <publisher><name>Morgan Kaufmann</name></publisher>
</book>
```
The answer to the previous query is:

<result>
  <author><lastname>Leighton</lastname></author>
  <title> An Introduction to Parallel Algorithms and Architectures </title>
</result>

<result>
  <author><lastname>Ceri</lastname></author>
  <title> Active Database Systems </title>
</result>

<result>
  <author><lastname>Widom</lastname></author>
  <title> Active Database Systems </title>
</result>
Previous query does not return pure XML; So, add <answer>...

<answer>
where
<book>
  <publisher><name>Morgan Kaufmann</name></publisher>
  <title>$T</title>
  <author>$A</author>
</book>
in "www.a.b.c/bib.xml"
construct
<result>
  <author>$A</author>
  <title>$T</title>
</result>
</answer>
Processing Optional Elements with Nested Queries

Assume <price> tag in <book> element is optional;

where

in "www.a.b.c/bib.xml"
construct <result><booktitle> $T </result>	<bookprice> $P </result>

is not correct because pattern insists <price> be present.

where

<title> $T </title> in $B
construct <result><booktitle> $T </result>
    where <price> $P </price> in $B
        construct <bookprice> $P </result>
</result>
So, the result looks like:

```xml
<result><booktitle>...</booktitle></result>
<result><booktitle>...</booktitle>
    <bookprice>...</bookprice></result>
<result><booktitle>...</booktitle></result>
```
Grouping with nested queries

The bibliography data has several authors for each book.
Suppose we want to retrieve each author and all book titles
he/she has published.

construct
  <result>
    <author> $A </author>
    in "www.a.b.c/bib.xml",
    construct <title> $T$ </title>
  </result>
Binding Elements and Contents

Variables in XML-QL are bound to element content rather than element itself.

XML-QL provides syntactic sugar that allows one to bind to element.

element_as keyword:

where <book><publisher><name>Morgan Kaufmann</name></publisher></book>
    element_as $B in "abc.xml"
construct $B

$B is bound to element <book>...</book>

The XML-QL processor will translate the query into

Where <book>$T</book> in "abc.xml",
  <publisher>$T</publisher><name>Morgan Kaufmann</name></publisher> in $T
construct <book>$T</book>
content_as keyword: allows one to bind to content

where <book><publisher><name>Morgan Kaufmann</name></publisher></book>
    content_as $C in "abc.xml"
construct <result>$C</result>

$C is bound to content within <book></book>

The XML-QL processor will translate the query into

Where <book>$C</book> in "abc.xml"
    <publisher><name>Morgan Kaufmann</name></publisher> in $C
construct <result>$C</result>
Querying Attributes:

Get all book titles in French.

```xml
where <book language="French">
    <title/> element_as $T
</> in "abc.xml"
construct $T
```

Get all languages in the database:

```xml
where <book language=$L/> in "abc.xml"
construct <result> $L </result>
```

Notice an attribute value in the XML document becomes an element value of the output of the query.
Joining Elements by Value:

By using the same variable in two matchings, we can express "joins"

Get all authors who have published at least two books.

```xml
where <book><author> $A </author></book>
    content_as $B1 in "abc.xml",
    <book> <author> $A </author></book>
    content_as $B2 in "abc.xml",
    B1 != B2
construct <result> $A </result>
```
Tag Variables:

Find all publications published in 1995 with Smith as an author or editor.

where <$P> <title> $T </title>
  <year> 1995 </>
  <$E> Smith </>
</ in "www.a.b.c/bib.xml",
$E in {author, editor}
construct <$P> <title> $T </title>
  <$E> Smith </>
</>

Here there are two Tag Variables: $P and $E;
$P is bound to top level tag (e.g. book, article, ...) and
$E is bound either to editor or author.
Regular Path Expressions

Consider the DTD that defined a self recursive element "part"

<!ELEMENT part(name, brand, part*)>
<!ELEMENT name (PCDATA)>
<!ELEMENT brand (PCDATA)>

To query such structures, XML-QL provides regular path expressions.

Get names of every part element that contains a brand element equal to "Ford" regardless of the nesting level.

where
   <part*> <name> $R </name>
       <brand> Ford </brand>
   </> in "www.a.b.c/x.xml"
construct <result> $R </>

The expression in the where clause above corresponds to the union of the following infinite sequence of patterns:

<name> $R$ </> <brand> Ford </>
<part><name>$R$</><brand> Ford </></>
<part><part><name>$R$</><brand> Ford </></></>
<part><part><name>$R$</><brand> Ford </></></></>
...
The wild card $ matches any tag and can appear anywhere a tag is permitted. For example:

```xml
where <$*> <name>$R</name><brand>Ford</brand><end>
in "www.a.b.c/x.xml"
construct <result>$R</result>
```

$* indicates a tag at any level (is abbreviated to *.)

So, `<*.brand>Ford</brand>` matches brand Ford at any level/depth in the XML graph.

```xml
where <part+.subpart+component.piece> $R </component.piece>
in "www.a.b.c/x.xml"
construct <result> $R </result>
```
Order: Skip this section
XSL: XML Stylesheet Language

- XSL primarily allows users to write transformations from XML to HTML thus describing the presentation.

- Can also be used as a transformation language in data exchange applications.

- XSL program consists of a set of TEMPLATE rules.

- Each rule consists of a pattern and a template.
  pattern => where clause; template => construct clause

- XSL processor starts from the root element and tries to apply a pattern to that node;

- If it succeeds, it executes the corresponding template.

- The template, when executed, usually instructs the processor to produce some XML result and to apply the templates recursively on the node’s children.

- XSL program is like a recursive function.
Sample XML data:

```xml
<bib>
  <book>
    <title>t1</title>
    <author>a1</author>
    <author>a2</author>
  </book>
  <paper>
    <title>t2</title>
    <author>a3</author>
    <author>a4</author>
  </paper>
  <book>
    <title>t3</title>
    <author>a5</author>
    <author>a6</author>
    <author>a7</author>
  </book>
</bib>
```
The following XSL program returns all titles:

```xml
<xsl:template>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="/bib/*title">
  <result>
    <xsl:value-of/>
  </result>
</xsl:template>
```

- When a match attribute is missing, the template matches any node.
- The match attribute defines the pattern (unix-like path names)
  (Here it is bib followed by any label followed by title)
- `<xsl:value-of/>` means value of current node (its content)
- The first rule is applied; It matches the root node;
  The template tells the XSL processor to recursively
  apply the templates on the children nodes `<book>...</book> etc.`
- Recursively, this proceeds until at the third level, the second
  rule applies and a result is produced.

The result will have the form:
```xml
<result> t1 </result>
<result> t2 </result>
<result> t3 </result>
```
XSL Patterns:

- bib matches a bib element
- * matches any element
- / matches the root
- /bib matches a bib element immediately after the root
- bib/paper matches paper following a bib
- bib//paper matches a paper following a bib at any depth
- //paper matches a paper at any depth
- paper|book matches a paper or book
- @language matches a language attribute
- bib/book/@language matches language attribute of a book which follows bib

XSL data model assumes a root node on top of the top element of the XML data model
/ node is on top of bib node

This is to accommodate processing statements before the top node is encountered. For example:

<!-- comment 1 -->
<!-- comment 2 -->
<bib>...</bib>
<!-- comment 3 -->

From XSL's view point the root node for above data has 4 children: 3 comments and one <bib>...</bib> element.
Non-linear patterns using [] notation:

```
paper[year] matches a paper element which has a year subelement
paper/year matches a year element which appears as a sub-element of paper.

bib/paper[year and publisher/name and @language]

matches a paper element in bib but only if it has a year sub-element, a publisher sub-element and a language attribute.
```

XSL does not allow variables; a limitation.
XSL Template Rules:

General Form:

```xml
<xsl:template match="pattern">
    template
</xsl:template>
```

XSL processor automatically prepends // to each pattern;
So, /book/*/title can be abbreviated to title

The template consists of XML (or HTML) code along with
XSL instructions;
XSL instructions:
<xsl:value-of/> evaluates to the string content of the current node.
<xsl:element name="..."> ... <xsl:element/> creates a new element

<xsl:template match="A">
  <xsl:element name="B">
    <xsl:value-of/>
    <xsl:element/>
  </xsl:element>
</xsl:template>

is equivalent to

<xsl:template match="A">
  <B> <xsl:value-of/> </B>
</xsl:template>
A situation where `<xsl:element>` is truly useful is the following which copies all top-level elements from the input file:

```xml
<xsl:template match="*">
  <xsl:element name="name()">
    <xsl:value-of/>
  </xsl:element>
</xsl:template>
```

`name()` returns the name of the current node which we use as the name of the output node.
Example of generating HTML from XML document:

```xml
<xsl:template match="/">
  <HTML>
    <HEAD>
      <TITLE>Bibliography Entries</TITLE>
    </HEAD>
    <BODY>
      <xsl:apply-templates/>
    </BODY>
  </HTML>
</xsl:template>

<xsl:template match="title">
  <TD>
    <xsl:value-of/>
  </TD>
</xsl:template>

<xsl:template match="author">
  <TD>
    <xsl:value-of/>
  </TD>
</xsl:template>

<xsl:template match="book|paper">
  <TR>
    <xsl:apply-templates select="title"/>
    <xsl:apply-templates select="author"/>
  </TR>
</xsl:template>
```
Continued:

```xml
<xsl:template match="bib">
  <TABLE>
    <TBODY>
      <xsl:apply-templates/>
    </TBODY>
  </TABLE>
</xsl:template>

<HTML>
  <HEAD>
    <TITLE>Bibliography Entries</TITLE>
  </HEAD>
  <BODY>
    <TABLE>
      <TBODY>
        <TR><TD> t1 </TD> <TD> a1 </TD> <TD> a2 </TD> </TR>
        <TR><TD> t2 </TD> <TD> a3 </TD> <TD> a4 </TD> </TR>
        <TR><TD> t3 </TD> <TD> a5 </TD> <TD> a6 </TD> <TD> a7 </TD> </TR>
      </TBODY>
    </TABLE>
  </BODY>
</HTML>
```