CSE 391	Web Queries: Methods and Tools	Handout MP
Spring 2004	Annie Liu	Mar. 4, 2004
Stony Brook	**Midterm Practice Problems**	No Due Date

1. Specify a DTD appropriate for a document that contains data from both the Course table in Figure 5.15, page 116 (of the big textbook), and the Requires table in Figure 5.16, page 117. Try to reflect as many constraints as the DTDs allow. Give an example of a document that conforms to your DTD.

One good example of such a document is shown below. Note that it does not try to mimic the relational representation, but instead courses are modeled using nested structures (i.e., the object-oriented approach).

```
<Courses>
  <Course CsrCode="CS315" DeptId="CS"
          CrsName="Transaction Processing" CreditHours="3">
    <Prerequisite CrsCode="CS305" EnforcedSince="2000/08/01"/>
    <Prerequisite CrsCode="CS219" EnforcedSince="2001/01/01"/>
  </Course>
  <Course>
    .....
  </Course>
  .....
</Courses>
```

An appropriate DTD would be:

```
<!DOCTYPE Courses [
  <!ELEMENT Courses (Course*)>
  <!ELEMENT Course (Prerequisite*)>
  <!ELEMENT Prerequisite EMPTY>
  <!ATTLIST Course
                CrsCode ID #REQUIRED
                DeptId CDATA #IMPLIED
                CrsName CDATA #REQUIRED
                CreditHours CDATA #REQUIRED >
  <!ATTLIST Prerequisite
                CrsCode IDREF #REQUIRED
                EnforcedSince CDATA #REQUIRED>
]>
```

2. Define the following simple types:
(a) A type whose domain consists of lists of strings, where each list consists of 7 elements.

```
<simpleType name="ListsOfStrings">
  <list itemType="string" />
</simpleType>
<simpleType name="ListsOfLength7">
  <restriction base="ListsOfStrings">
    <length value="7"/>
  </restriction>
</simpleType>
```

(b) A type whose domain consists of lists of strings, where each string is of length 7.

```
<simpleType name="StringsOfLength7">
  <restriction base="string">
    <length value="7"/>
 </restriction>
</simpleType>
```

```
<simpleType name="ListsOfStringsOfLength7">
  <list itemType="StringsOfLength7"/>
</simpleType>
```

(c) A type appropriate for the letter grades that students receive on completion of a course—A, A-, B+, B, B-, C+, C, C-, D, and F. Express this type in two different ways: as an enumeration and using the pattern tag of XML Schema.

```
<simpleType name="gradesAsEnum">
  <restriction base="string">
    <enumeration value="A"/>
    <enumeration value="A-"/>
    <enumeration value="B+"/>
    ..........
  </restriction>
</simpleType>
<simpleType name="gradesAsPattern">
  <restriction base="string">
    <pattern value="(A-?|[BC][+-]?|[DF])"/>
  </restriction>
</simpleType>
```

In the gradesAsPattern representation, (...) represent complex alternatives of patterns separated by — (W3C has adopted the syntax of regular expressions used in Perl), [...] represent simple alternatives (of characters), and ? represents zero or one occurrence, as usual in regular expressions.

3. Write an XML schema specification for a simple document that lists stock brokers with the accounts that they handle and a separate list of the client accounts. The Information about the accounts includes the account Id, ownership information, and the account positions (i.e., stocks held in that account). To simplify the matters, it suffices, for each account position, to list the stock symbol and quantity. Use ID, IDREF, and IDREFS to specify referential integrity.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:brk="http://somebrokerage.com/documents"
        targetNamespace="http://somebrokerage.com/documents">
  <element name="Brokerage">
    <complexType>
      <sequence>
        <element name="Broker" type="brk:brokerType"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element name="Account" type="brk:accountType"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
  <complexType name="brokerType">
    <attribute name="Id" type="ID"/>
    <attribute name="Name" type="string"/>
    <attribute name="Accounts" type="IDREFS"/>
  </complexType>
  <complexType name="accountType">
    <attribute name="Id" type="ID"/>
    <attribute name="Owner" type="string"/>
    <element name="Positions"/>
      <complexType>
        <sequence>
          <element name="Position">
            <complexType>
```

```
            <attribute name="stockSym" type="string"/>
            <attribute name="qty" type="integer"/>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
 </complexType>
</schema>
```

4. Formulate the following XPath queries for the document in Figure 18.21, page 713:
(a) Find the names of all courses taught by Mary Doe in fall 1995.

```
//CrsName[../Instructor="Mary Doe" and ../@Semester="F1995"]/text()
```

Note that we use text() at the end to get the text (course names themselves) as opposed to CrsCode element nodes.
(b) Find the set of all document nodes that correspond to the course names taught in fall 1996 or all instructors who taught MAT123.

```
//CrsName[../@Semester="F1996"] | //Instructor[../@CrsCode="MAT123"]
```

(c) Find the set of all course codes taught by John Smyth in spring 1997.

```
//Class[Instructor="John Smyth" and @Semester="S1997"]/@CrsCode
```

5. Use XSLT to transform the document in Figure 18.19, page 710, into a well-formed XML document that contains the list of Student elements such that each student in the list took a course in spring 1996.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
    <StudentListS1996>
      <xsl:apply-templates/>
    </StudentListS1996>
  </xsl:template>
  <xsl:template match="//Student">
    <xsl:if test="../CrsTaken/@Semester='S1996'">
      <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

6. Write an XSLT stylesheet that traverses the document tree and, ignoring attributes, copies the elements and doubles the text nodes. For instance,`<foo a="1">the<best/>bar</foo>` would be converted into `<foo>thethe<best/>barbar</foo>`.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/|*">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="text()">
    <xsl:value-of select=".">
    <xsl:value-of select=".">
  </xsl:template>
</xsl:stylesheet>
```

7. Use the document structure described in exercise 18.21 to formulate the following queries in XQuery:
(a) List all classes (identified by a course code and semester) where every student received a B or higher.

```
DECLARE FUNCTION extractClasses ($t AS element()) AS element()* {
  <Class CrsCode=$t/CrsCode Semester=$t/Semester/>
}
<EasyClasses>
{
  FOR $c IN
        distinct-values(FOR $t IN document("http://xyz.edu/transcript.xml")//tuple
                        RETURN extractClasses($t))
  LET $grades :=
        document("http://xyz.edu/transcript.xml")
        //tuple[CrsCode/@value=$c/CrsCode/@value and
                Semester/@value=$c/Semester/@value]
        /Grade/@value
  WHERE EVERY $g IN $grades
        SATISFIES numericGrade($g) >= numericGrade("B")
  RETURN $c
}
</EasyClasses>
```

(b) List all students who never received less than a B.

```
<GoodStudents>
{
  FOR $sid IN
        distinct-values(document("http://xyz.edu/transcript.xml")//tuple/StudId)
      $s IN document("http://xyz.edu/student.xml")
            //tuple[Id/@value=$sid/StudId/@value]
  LET $grades :=
        document("http://xyz.edu/transcript.xml")
        //tuple[StudId/@value=$sid/StudId/@value]/Grade/@value
  WHERE EVERY $g IN $grades
        SATISFIES numericGrade($g) >= numericGrade("B")
  RETURN $s
}
</GoodStudents>
```

8. Write an XQuery function that traverses a document and computes the maximum branching factor of the document tree, i.e., the maximal number of children (text or element nodes) of any element in the document.

```
DECLARE NAMESPACE xsd = "http://www.w3.org/2001/XMLSchema"

DECLARE FUNCTION maxBranching($n) AS xsd:integer {
  LET $children := $n/node()
  LET $maxChildBranching := max{
        FOR $m IN $children
        RETURN maxBranching($m)
      }
  LET $childCount := count($children)
  RETURN
        IF $childCount >= $maxChildBranching
        THEN $childCount
        ELSE $maxChildBranching
}
```

Recall that the XPath function node() matches every e- or t-node.