XML

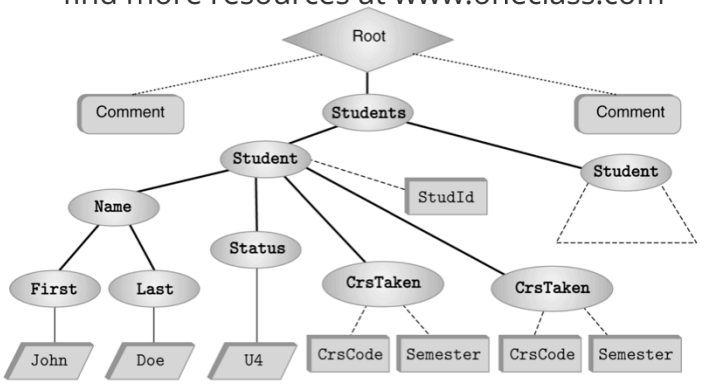
- Data Types:
  - **Structured Data** – well-known data format
    - Ex. Databases w/ relations and tuples, conforms to schema
  - **Unstructured Data** – cannot assume any predefined format
    - Apparent organization makes no guarantees
    - Self-describing: little external knwoeldge needed, but need to infer what the data means
    - Ex. Plan Text
  - **Semistructured Data** – enforce "well-formatted" data
    - Known how to read/parse/manipulate it
    - Optionally, enforce "well-structured" data, might help interpret
    - Pro: Highly protable; Con: verbose/redundant
    - Ex. XML, HTML
- **XML** – designed for data interchange
  - Features of XML:
    - **Tree-structured** (hierarchical) format w/ nested elements
    - **Elements** surrounded by opening and closing **tags**
    - **Attributes** embedded in opening tags
    - **Strictly well-formed** → must close all tags, etc.
    - Tag/attribute names carry **no semantic meanin**g
    - Data-only format → **no implied presentation**
    - Names (Elements, Attributes) must be valid identifiers: [a-z][A-Z]
  - Example XML:

```
1.  <?xml version="1.0" ?>
2.  <PersonList type="Student" date="2002-02-02">
3.      <Title value="Student List" />
4.      <Person name="John" Id="s111111111">
5.          John is a nice fellow
6.          <Address>
7.              <Number>21</Number>
8.              <Street>Main St.</Street>
9.          </Address>
10.     </Person>
11.     <Person>
12.         ...
13.     </Person>
14. </PersonList>
```

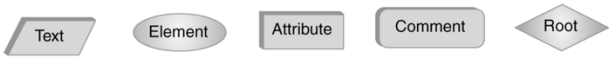
    - Notes:
      - **Root Element** opening tag on line 2, closing tag on line 14 and contains all the other elements nested inside
      - Root element has attributes type, and date
      - **Empty Element** = Title element (line 3), has no children
      - **Standalone Text** = line 5, not useful as data, non-uniform
      - Nested **Child Element Address** (lines 6 – 9) inside **Parent Element** Person (lines 4 – 10)
      - Element Street (line 8) is a **Child Element** of Address, and a **Descendant Element** of Person
  - Example 2:

```
1.  <?xml version="1.0" ?>
2.  <!-- Some comment -->
3.  <Students>
4.      <Student StudId="111111111" >
5.          <Name>
6.              <First>John</First>
7.              <Last>Doe</Last>
8.          </Name>
9.          <Status>U2</Status>
10.         <CrsTaken CrsCode="CS308"
11.                   Semester="F1997" />
12.         <CrsTaken CrsCode="MAT123"
13.                   Semester="F1997" />
14.     </Student>
15.     <Student StudId="987654321" >
16.         <Name>
17.             <First>Bart</First>
18.             <Last>Simpson</Last>
19.         </Name>
20.         <Status>U4</Status>
21.         <CrsTaken CrsCode="CS308"
22.                   Semester="F1994" />
23.     </Student>
24. </Students><!-- Some other comment -->
```


  - Resultant XML Tree:



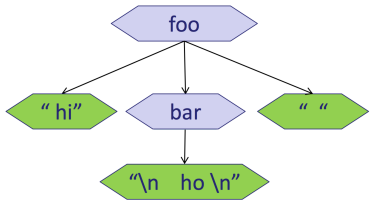
    - Legend:



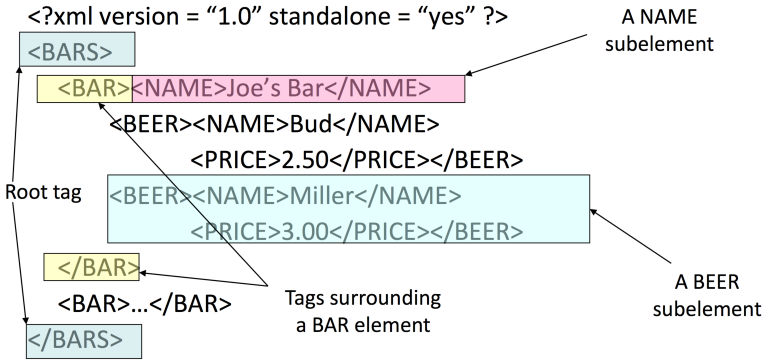
- XML Type 1: **Well-formed XML**:
  - Can invent tags
  - Any tag can go anywhere
  - Rules for well-formed XML:
    - 1. Must have a **root element**
    - 2. Every **opening tag** must have matching **closing tag**
    - 3. Elements must be **properly nested**:
      - ex. Improper nesting: `<foo><bar></foo></bar>`
    - 4. An **attribute** name can occur at **most once** in an opening tag, and if it occurs:
      - 1. It must have **explicity specified value** (booleans are not allowed)
      - 2. The value must be **quoted** (with single/double quotes)
  - Text and Whitespace
    - Parser never ignores whitespace
      - Leading/trailing space befire/after tags → text nodes
      - White space btwn tags → empty text nodes
    - Example:

```
1.  <foo> hi<bar>
2.    ho
3.  </bar> </foo>
```

      - Resulting tree:



  - Example Well-formed XML:



  - Drawbacks of Well-formed XML, missing information about
    - What tags are allowed
    - What order, nesting
    - What attributes for each tag
    - What's mandatory or optional

- XML Type 2: **Valid XML with Document Type Definition (DTD)**
  - Document Type Definition
    - Enforces beyond "well-formed"-ness
      - Which elements may (must) appear where
      - Attributes the elements may (must) possess
      - Types attributes and data must adhere to
    - DTD is separate from the XML document it constriants
    - Entire DTD nested within a !DOCTYPE tag, given root element name:
      `<!DOCTYPE ROOT-NAME [...]>`
  - DTD **Elements**: `<!ELEMENT $e content-req>`
    - $e = name of the element tag
    - content-req is the requires given to the content of this tag name
      - No content → `<!ELEMENT $e EMPTY>`
      - Anything → `<!ELEMENT $e ANY>`
      - Text Data → `<!ELEMENT $e (#PCDATA)>`
      - Children → `<!ELEMENT $e (...)>`
      - Mixed → `<!ELEMENT $e (#PCDATA|...)>`
    - Text Types:
      - PCDATA: Parsed character data –i.e. mixed text and markup
      - CDATA: Non-parsed character data –i.e. plain text
  - DTD **Children Elements**:
    - **A sequence**: `<!ELEMENT $e (a,b,c)>`
      - Comma defines order which children must appear in XML
    - **Either-Or**: `<!ELEMENT $e (a|b|c)>`
      - Exactly one of the options must appear in the XML
    - **Constrain Cardinality**: `<!ELEMENT $e (a,b+,c*,d?)>`
      - No suffix = exactly 1
      - + suffix = at least one, or more
      - * suffix = zero, or more
      - ? suffix = at most one
    - Sequences and either-or can be nested
    - Example1:
      - Sequence:

```
<!ELEMENT resume (
    bio, interests, education,
    exper, awards, service)>
```

      - Sequence /w constraints:

```
<!ELEMENT bio (
    name, address, phone,
    email?, fax?, url?)>
<!ELEMENT interests (interest+)>
<!ELEMENT education (degree*)>
```

      - Nested

```
<!ELEMENT awards ((award|honor)*)>
```

    - Example2:

```
<!DOCTYPE BARS [
   <!ELEMENT BARS (BAR*)>
   <!ELEMENT BAR (NAME, BEER+)>
   <!ELEMENT BEER (NAME, PRICE)>
   <!ELEMENT NAME (#PCDATA)>
   <!ELEMENT PRICE (#PCDATA)>
]>
```

      - Bars element has zero or more Bar element nested within
      - Bar element has one Name element and one or more Beer elements
      - Beer element has one Name element and one Price Element
      - Name and Price are texts
  - DTD **Attributes**: `<!ATTLIST $e $a $type $required>`
    - **$e** = the element associated with the list of attributes **$a**
    - **$type** may be any of the following:
      - Character Data: CDATA
      - One of a set of values: (v1|v2|...)
      - Unique Identifier: ID
      - ID constrains: ID[REF[S]]
      - Valid XML name (or list of names): NMTOKEN[S]
      - Entity/Entities: ENTITY/ENTITIES
    - ID Constraints:
      - **ID** = uniquely identifies an element in document, error to have more than 1
      - **IDREF** = references another element by its ID attribute, error if given ID doesn't exists
      - **IDREFS** = list of IDREF attributes, space-seperated
    - **$required** may be any of the following
      - Required: #REQUIRED
      - Not Required: #IMPLIED
      - Fixed Value (always same): #FIXED "$value"
      - Default Value: "$value"
      - Note: #IMPLIED is assigned unless specified otherwise
    - Example 1:

```
<!ATTLIST person sin ID #REQUIRED>
<!ATTLIST person spouse IDREF #IMPLIED>
<!ATTLIST person name CDATA "John Doe">
<!ATTLIST person trusted (yes|no) "no">
<!ATTLIST person species #FIXED "human">
<!ATTLIST person alive (yes|no) #IMPLIED>
```

    - Example 2:
      - The DTD: `bars.dtd`

```
<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*, BEER*)>
    <!ELEMENT BAR (SELLS+)>
        <!ATTLIST BAR name ID #REQUIRED>
    <!ELEMENT SELLS (#PCDATA)>
        <!ATTLIST SELLS theBeer IDREF
                        #REQUIRED>
    <!ELEMENT BEER EMPTY>
        <!ATTLIST BEER name ID #REQUIRED>
        <!ATTLIST BEER soldBy IDREFS
                       #IMPLIED>
]>
```

      - The XML Document: `bars.xml`

```
1.  <?xml version="1.0">
2.  <!DOCTYPE BARS "./bars.dtd">
3.
4.  <BARS>
5.      <BAR name="JoesBar">
6.          <SELLS theBeer="Bud">2.50</SELLS>
7.          <SELLS theBeer="Miller>3.00</SELLS>
8.      </BAR>
9.      <BAR name="SuesBar">
10.         <SELLS theBeer="Bud">2.50</SELLS>
11.     </BAR>
12.
13.     <BEER name="Bud"
14.           soldBy="JoesBar SuesBar"/>
15.     <BEER name="Miller" soldBy="JoesBar"/>
16. </BARS>
```

  - DTD **Entities**: `<!ENTITY $name "$substitute-val">`
    - Define an entity with name, which has the value substitute-val, ex:

```
<!ENTITY buzz-word "communism">
<!ENTITY buzz-word "racism">
<!ENTITY buzz-word "terrorism">
<!ENTITY buzz-word "illegal file sharing">
```

    - Use:

```
<political-speak>
    I vow to lead the fight to stamp out
    &buzz-word; by instituting powerful new
    programs that will...
</political-speak>
```

    - Predefine entities:
      - Quotation Mark " &quot
      - Apostrophe ' &apos
      - Greater Than > &gt
      - Lesser Than < &lt
      - Ampersand & &amp

- DTD types
  - Embedded DTD: specified as part of the XML document; ex.
    - If any elements, attributes, or entities are used in the XML document that are referenced or defined outside the current document → Standalone = no; otherwise Standalone = yes

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Book [
    ...
]>
<Book> ... </Book>
```

    - Example:

```
<?xml version="1.0" standalone="no" ?>

<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*)>
    <!ELEMENT BAR (NAME, BEER+)>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT BEER (NAME, PRICE)>
    <!ELEMENT PRICE (#PCDATA)>
]>

<BARS>
    <BAR>
        <NAME>Joe's Bar</NAME>
        <BEER>
            <NAME>Bud</NAME>
            <PRICE>2.50</PRICE>
        </BEER>
        <BEER>
            <NAME>Miller</NAME>
            <PRICE>3.00</PRICE>
        </BEER>
    </BAR>
    <BAR> ... </BAR>
</BARS>
```

  - Reference external DTD document; ex.

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Book "some-url/book.dtd">
<Book> ... </Book>
```

    - Ex. The DTD: `bars.dtd`

```
<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*)>
    <!ELEMENT BAR (NAME, BEER+)>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT BEER (NAME, PRICE)>
    <!ELEMENT PRICE (#PCDATA)>
]>
```

    - Ex. The XML Document: `bars.xml`

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Book "./bars.dtd">

<BARS>
    <BAR>
        <NAME>Joe's Bar</NAME>
        <BEER>
            <NAME>Bud</NAME>
            <PRICE>2.50</PRICE>
        </BEER>
        <BEER>
            <NAME>Miller</NAME>
            <PRICE>3.00</PRICE>
        </BEER>
    </BAR>
    <BAR> ... </BAR>
</BARS>
```

  - DTD Limitations
    - No namespaces, other than global
    - Very limited number of types
    - Very weak referential integrity
      - ID, IDREF, IDREFS use global ID space
    - Can't express unordered contents conveniently
      - Ex. attributes a, b, c must all appear but in any order
    - All element names are global
      - Must use name-company and name-person to differentiate between different name tags
- XML Schema (not covered)
  - Improve on DTDs
  - Pros:
    - Integrated w/ namespaces
    - Many built-in types
    - User-defined types
    - Has local element names
    - Powerful key and referential constraints
  - Cons:
    - Unwieldy, more complex than DTDs

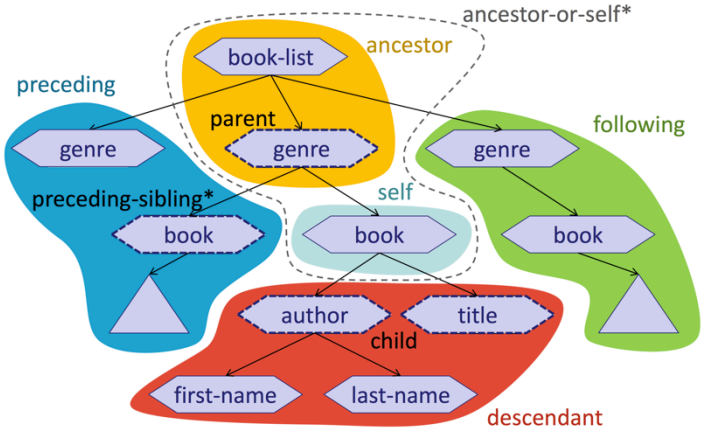
XPATH

- **XPath** = Query over hierarchical data
  - Recall: all XML documents are trees
    - Therefore, each element has a unique path to the root element
  - Idea: specify full or partical paths, w/ selection
    - ex. suppose following XML

```
1. <?xml version="1.0" standalone="no" ?>
2. <book-list>
3.     <book>
4.         <title>...</title>
5.         ...
6.     </book>
7.     ...
8. </book-list>
```

      - return the title of each book in book-list:
        `/book-list/book/title`
      - return any book whose title is "SQL for Dummies"
        `/book-list/book[title="SQL for Dummies"]`

  - **Axis** = what "dimension" of the tree to consider next
    - **Context Node** = current position in the XML tree
    - Element Axes
      - **`self`**: selects the Context Node
      - **`parent`**: Selects all parent of the Context Node
      - **`ancestor`**: selects all ancestors (parent, parent-parent, etc.) of Context Node; ex match all ancestors.
      - **`ancestor-or-Self`**: selects all ancestors of Context Node and Context Node itself
      - **`child`**: selects all children of the Context Node; Ex. Return the Child Nodes of the Context Node
      - **`descendant`**: Selects all descendants (children, children-children) of Context Node
      - **`descendant-or-self`**: Selects all descendants of Context Node and Context Node itself
      - **`following`**: selects everything in the document after the closing tag of the Context Node
      - **`following-sibiling`**: selects all siblings are the Context Node
      - **`preceding`**: Selects all nodes that appear before the Context Node in the document, except ancestors
      - **`preceeding-sibiling`**: Selects all sibilings before the Context Node



    - Non-element axes
      - **`attribute`**: retrieve attributes of the context node
      - **`namespace`**: retrieve node namespace
    - Selecting elements vs. text
      - `axes::*` → selects child elements only
      - `axes::text` → selects text children only
      - `axes::node()` → selects everything (child, and text)
    - Element position (in document order, starts from 1)
      - `elem::position()` → returns position of elem
      - `elem::last()` → returns num of nodes in the elem
    - Shortforms
      - `*` → all elements of current axis
      - `.` → `self::node()`
      - `..` → `parent::node()`
      - `elem_name` → `child::elem_name`
      - `@attr_name` → `attribute::attr_name`
      - `//` → `descendant-or-self::node()/`
      - `[3]` → `[position()=3]`
      - `[last()]` → `[position()=last()]`
  - **Path expression**
    - Absolute Path, following are the same
      - Ex. `/book-list/genre`
      - Ex. `/book-list/genre/self::node()`
    - Relative Path, following are the same
      - Ex. `book/title`
      - Ex. `./book/title/parent::*/child:: *`
    - Search any path use: `//`
  - **Predicates**
    - `[$expr]` applieds boolean predicate to a node set (similar to an if)
      - returns subset of nodes $expr is TRUE, exclude others
    - Expresion can contain:
      - Boolean constants → `true()`, `false()`
      - Numbers → false for `0`, or `NaN`
      - Strings → false for empty string
      - Comparisons → (=, !=, <, >, etc.)
      - Compound → $expr1 and $expr2 or $expr3
    - Can reference child element:
      - Ex. `//book[title!="Harry Potter"]`
    - Can reference attributes of current element:
      - Ex. `book-list/book[@special-offer]`
  - Nesting Paths and Predicates:
    - Chain path steps and/or predicates
      - Ex. `/book-list/genre/book[price < 50][1]`
      - Ex. `/book-list/genre/book[price < 50]`
        `/author[last-name='Asimov']`
    - Full nesting, the following are equivalent
      - Ex. `/book-list/genre`
        `/book[author[last-name='Asimov']]`
      - Ex. `/book-list/genre`
        `/book[author/last-name='Asimov']`
  - Parenthesis: combine results of 2+ XPath queries
    - Ex. titles and publisher of all books written by Issac Asimov

```
//book[author/last-name="Asimov"]
                        /(title|publisher)
```

    - Ex. books whos keyword or title mentions "robot"

```
//book[(keyword|title)
          [contains(text(), "robot")]]
```

  - Functions:
    - Node-related
      - `count($node-set)` → cardinality of $node-set
      - `id($idarg)` → returns element of specified ID
      - `name($node-set)` → tag name of $node-set[1]
    - Number-related
      - `number($arg?)` convert $arg or . into a number
      - `sum($node-set)` converts the nodes to numbers and sums them
      - `floor`, `round`, `ceil`
    - String manipulation
      - `string($arg?)` converts $arg or . into a string
      - `starts-with($str, $prefix)`
      - `contains($haystack, $needle)`
      - `substring($str, $beg, $len?)`
      - `normalize-space($arg?)` removes \n and \t in $arg
      - `string-length($arg?`
      - `concat($s1,$s2,...)`

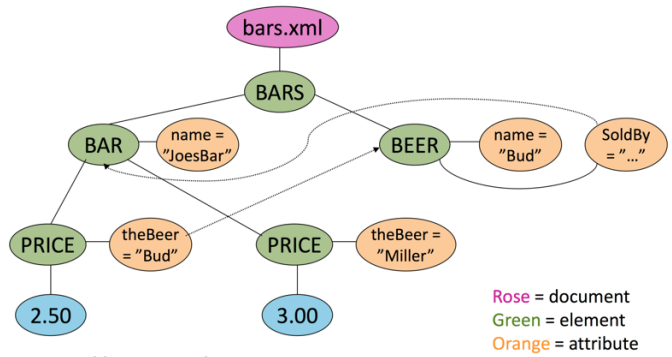
- XPath results:
  - A query never modifies nodes
  - Predicates "marks" nodes
    - Predicates cannot cause a node to be output twice
    - Returning a node returns all of its children
  - Output produced in document order
    - Ordering preserved across unions
- Example:
  - The DTD file: `bars.dtd`

```
<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*, BEER*)>
    <!ELEMENT BAR (PRICE+)>
        <!ATTLIST BAR name ID #REQUIRED>
    <!ELEMENT SELLS (#PCDATA)>
        <!ATTLIST SELLS theBeer IDREF #REQUIRED>
    <!ELEMENT BEER EMPTY>
        <!ATTLIST BEER name ID #REQUIRED>
        <!ATTLIST BEER soldBy IDREFS #IMPLIED>
]>
```

  - The XML file

```
1.  <?xml version="1.0" standalone="no" ?>
2.  <!DOCTYPE Book "./bars.dtd">
3.
4.  <BARS>
5.      <BAR name="JoesBar">
6.          <PRICE theBeer="Bud">2.50</PRICE>
7.          <PRICE theBeer="Miller">3.00</PRICE>
8.      </BAR>
9.      <BAR name="SuesBar">
10.         <PRICE theBeer="Miller">3.50</PRICE>
11.     </BAR>
12.
13.     <BEER name="Bud"
14.           soldBy="JoesBar SuesBar"/>
15.     <BEER name="Miller" soldBy="JoesBar"/>
16. </BARS>
```

    - Tree Representation



Horizontal lines: attributes
Non-horizontal solid lines: nesting of elements
Dashed lines: from id/idrefs

  - **Query 1**: `/BARS`
    - Result:

```
1.  <BARS> ... </BARS>
```

  - **Query 2**: `/BARS/BAR`
    - Result:

```
1.  <BAR name="JoesBar"> ... </BAR>
2.  <BAR name="SuesBar"> ... </BAR>
```

  - **Query 3**: `/BARS/BAR/PRICE`
    - Result:

```
1.  <PRICE theBeer="Bud">2.50</PRICE>
2.  <PRICE theBeer="Miller">3.00</PRICE>
3.  <PRICE theBeer="Miller">3.50</PRICE>
```

  - **Query 4**: `/BARS/BAR/PRICE/@theBeer`
    - Result:

```
1.  theBeer="Bud"
2.  theBeer="Miller"
3.  theBeer="Miller"
```

  - **Query 5**: `//PRICE`
    - Result:

```
1.  <PRICE theBeer="Bud">2.50</PRICE>
2.  <PRICE theBeer="Miller">3.00</PRICE>
3.  <PRICE theBeer="Miller">3.50</PRICE>
```

  - **Query 6**: `/BARS/*`
    - Result:

```
1.  <BAR name="JoesBar"> ... </BAR>
2.  <BAR name="SuesBar"> ... </BAR>
3.  <BEER name="Bud"
          soldBy="JoesBar SuesBar"/>
4.  <BEER name="Miller" soldBy="JoesBar"/>
```

  - **Query 7**: `/BARS/BAR/PRICE[. < 2.75]`
    - Notice: . gets the value of the current PRICE element
    - Result:

```
1.  <PRICE theBeer="Bud">2.50</PRICE>
```

  - **Query 8**: `/BARS/BAR/PRICE[@theBeer="Miller"]`
    - Result:

```
1.  <PRICE theBeer="Miller">3.00</PRICE>
2.  <PRICE theBeer="Miller">3.50</PRICE>\
```

- XPath limitations
  - Most joins impossible
  - No means of formatting results
    - XPath query returns a list of elements
      - Might want to wrap those elements inside elements
    - XPath can only return full subtrees
    - Ex. Query: book_list[book/author/last-name="Asimov"]
      - Returns a list of complete book elements
      - Want:

```
1.  <book_list>
2.      <book>
3.          <title>I, Robot</title>
4.          <publisher>Gnome Press</publisher>
5.      </book>
6.      ...
7.  </book_list>
```

  - No control flow (branching or loops)
  - Little/no ability to manipulate XML
  - No way to specify input

XQuery

- XQuery features:
  - Templates: mixed output and logic
    - Statically create the overall structure
    - Embed logic to handle input data
  - All expressions return XML
    - Outputs of one operation can be input to another
    - Returned value may be text, element, or node set
  - FLWOR expressions
    - Allow iteration over node sets and other sequences
  - Functions,
    - Allow logic encapsulation, recursion
- XQuery code format
  - Ex. simple XQuery Code
    - Static Template = lines 1, 2, 6, 7, 11
    - Interpreted Code = lines 3–5, 8–10

```
1.  <title>Information about Tor</title>
2.  <book-list>
3.  {
4.      //book[publisher="Tor"]/title
5.  }
6.  </book-list>
7.  <author-list>
8.  {
9.      //book[publisher="Tor"]/author/last-name
10. }
11. </author-list>
```

  - **FLWOR Expressions**
    - F = **For**
      - Iterate over each item in a sequence
      - Multiple sequences separated by commas
    - L = **Let**
      - Declares a variable and assigns it a value
      - Multiple declarations separated by commas
      - Bound once per iteration of every for above it
    - W = **Where**, O = **Order By**
      - Same as SQL
    - R = **Return**
      - The value that should be computed at each iteration
    - Output behavior:
      - In XPath, every node output at most once
      - In FLWOR, node output with every return
    - Example:
      - In XPath:

```
//book[publisher="Tor"
       and author/last-name="Asimov"]
       /(author|title)
```

      - In XQuery with FLWOR:

```
FOR $b in //book
LET $a := $b/author
WHERE $b/publisher="Tor"
    AND $a/last-name="Asimov"
ORDERBY $b/title
RETURN <book>{$b/title, $a}</book>
```

  - **Sequences**
    - Most expressions return sequences of nodes, ex.
      - $b is a sequence: `LET $b = /bib/book`
      - a sequence: `$b/@isbn`
      - a sequence of n prices: `$b/price`
      - a sequence of n nums: `$b/price*0.7`
      - a sequence of n*m nums: `$b/price*$b/quantity`
    - Sequence literals
      - $b is a a sequene of 1, 2, 3 `LET $b = (1, 2, 3)`
      - $b is a a sequene of 1–10 `LET $b = (1 to 10)`
    - Sequences can combine, but flatten
      - Ex. `(1, 2, (3, 4, 5), (), 6, 7)` → `(1, 2, 3, 4, 5, 6, 7)`
  - **If-Then-Else Expressions**
    - Syntax: `if ($expr) then $expr else $expr`
    - Ex.

```
FOR $b IN //book
RETURN
    IF ($b/publisher="Tor")
    THEN <book>{$b/(title|author)}</book>
    ELSE ()
```

  - **Predicates (using XPath)**
    - Test for empty/non-empty
    - Iteratre over their members (for) and applay a predicate
  - **Quantification**
    - Existential: `SOME $x IN $y SATISFIES $expr`
    - Universal: `EVERY $x IN $y SATISFIES $expr`
    - Ex.

```
//genre[SOME $n IN .//author/last-name
        SATISFIES $n="Asimov"]/@name
/transcript/semester[EVERY $m IN mark
        SATISFIES $m > 3.5]
/transcript/semester[SOME $m IN mark
        SATISFIES $m < 2.0]
```

  - **Comparisons**
    - General comparisons: apply to sequences of objects
      - Operators: `= != < > <= >=`
      - Apply atomization to both operands, allow for sequences
      - Note: if A op B is true → there's 1 pair a op b is true
      - Attempts to cast appropriate" "required type" for comparison to work
    - Value comparisons: apply to values of two objects
      - Operators: `eq ne lt gt le ge`
      - Apply atomization to both operands, fails if it's a sequence
      - Does not automatically case data into proper type
    - Node comparisons: apply to two nodes based on doc position
      - Operators: `is` << (pred) >> (following)
      - Note: is true if both sides are actually the same node
  - **Set operators**
    - Operators: `UNION INTERSECT EXCEPT`
      - Duplicate elimination: `distinct-values()`
      - All based on node comparisons, not values → attributes and children must also match (recursive)
    - Syntax: `LET $c := ($a/title UNION $b/title)`
  - **User-Defined Functions**
    - Arguments can be type
      - Parenthesis after type name
      - Default type: `item()`
      - Cardinality controlled: `+ ? *`
    - Function body is an expression to evaluate
    - Ex.

```
DECLARE FUNCTION count_nodes($e AS element())
AS integer {
    1 + sum(
        FOR $c IN $e/*
        RETURN count-nodes($c)
    )
};
```

- XQuery Example
  - XML Document: bib.xml

```
<bib>

    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming the Unix
               environment</title>
        <author>
            <last>Stevens</last>
            <first>W.</first>
        </author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author>
            <last>Abiteboul</last>
            <first>Serge</first>
        </author>
        <author>
            <last>Buneman</last>
            <first>Peter</first>
        </author>
        <author>
            <last>Suciu</last>
            <first>Dan</first>
        </author>
        <publisher>Morgan Kaufmann
                   Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and
               Content for DigitalTV</title>
        <editor>
            <last>Gerbarg</last>
            <first>Darcy</first>
            <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic
                   Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

  - **Query 1**: Find all book title publishe after 1990

```
FOR $x IN doc("bib.xml")/bib/book
WHERE $x/year > 1990
RETURN $x/title
```

    - Result:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming the Unix
environment</title>
<title>Data on the Web</title>
<title>The Economics of Technology and
Content for DigitalTV</title>
```

  - **Query 2**: list all books of authors published by Morgan Kaufmann by its author and title
    - Note: solution will return all books by those authors published by Morgan Kaufmann, including books not published by Morgan Kaufmann

```
FOR $a IN distinct-nodes(
    doc("bib.xml")/bib/book[
        publisher="Morgan Kaufmann Publishers"
    ]/author
)
RETURN <result>
    $a,
    FOR $t IN doc("bib.xml")/bib/book[
        author=$a
    ]/title
    RETURN $t
</result>
```

  - **Query 3**: Find books whose price is larger than average

```
LET $a=AVG(doc("bib.xml")/bib/book/price)
FOR $b in doc("bib.xml")/bib/book
WHERE $b/price > $a
RETURN $b/title
```

  - **Query 4**: Group by author names their frist ten books, for authors having written more than 10 books
    - Note: solution assumes authors have unique last names

```
FOR $a IN distinct_nodes(
    doc("bib.xml")/bib/author/lastname
)
LET $books := doct("bib.xml)//book[
    SOME $y IN author/lastname SATISFIES ($y=$a)
]
WHERE COUNT($books) > 10
RETURN <result>
    {$a}
    {$books[1 to 10]}
</result>
```

  - **Query 5a**: For each book, compare prices offered by Amazon and Barns (Using Joins in XQuery)

```
<books-with-prices>{
    FOR $a IN doc("amazon.xml")/book,
        $b IN doc("barns.xml")/book
    WHERE $b/@isbn = $a/@isbn
    RETURN <book>{
        {$a/title}
        <price-amazon>{
            $a/price
        }</price-amazon>
        <price-barns>{
            $b/price
        }</price-barns>
    }</book>
}</books-with-prices>
```

  - **Query 5b**: For each book, compare prices offered by Amazon and Barns (Using Outer Joins in XQuery)

```
<books-with-prices>{
    FOR $a IN doc("amazon.xml")/book
    RETURN <book>{
        {$a/title}
        <price-amazon>{$a/price}</price-amazon>
        {FOR $b IN doc("barns.xml")/book
         WHERE $b/@isbn = $a/@isbn
         RETURN <price-barns>{
            $b/price
         }</price-barns>}
    }</book>
}</books-with-prices>
```

- **Query 5C**: For each book, compare prices offered by Amazon and Barns (Using Full-outer Joins in XQuery)

```
LET $allISBNs := distinct-values(
    doc("amazon.xml")/book/@isbn
    UNION
    doc("bn.xml")/book/@isbn
)
RETURN <books-with-prices>{
    FOR $isbn IN $allISBNs
    RETURN <book>
        {
         FOR $a IN doc("amazon.xml")/book
         WHERE $a/@isbn = $isbn
         RETURN <price-amazon>{
            $a/price
         }</price-amazon>
        }
        {
         FOR $b IN doc("barns.xml")/book
         WHERE $b/@isbn = $isbn
         RETURN <price-barns>{
            $b/price
         }</price-barns>
        }
   </book>
}</books-with-prices>
```

- **Query 6**: Make a list of holdings, ordered by title. For jounrlas, include the editor, and for all other holdings, include the author

```
FOR $h IN //holding
ORDERBY $h/title
RETURN <holding>{
    IF $h/@type="Journal"
    THEN $h/editor
    ELSE $h/author
}</holding>
```

- **Query 7**: Find titles of books in which both "sailing" and "windsurfing" are mentioned in the same paragraph

```
FOR $b IN doc("bib.xml")//book
WHERE SOME $p in $b//para SATISFIES
    contains($p, "sailing)
    AND contains($p, "windsurfing")
RETURN $b/title
```

- **Query 8**: Find titles of books in which "sailing" is mentioned in every paragraph

```
FOR $b IN doc("bib.xml")//book
WHERE EVERY $p in $b//para SATISFIES
    contains($p, "sailing)
RETURN $b/title
```

- **Query 9:** Find the publishers and the books they have published order by publisher name and then by book price descending

```
<publisher_list>{
    FOR $p IN distinct-values(
        doc("bib.xml")//publisher
    )
    ORDERBY $p/name
    RETURN <publisher>{
        <name>$p/text()</name>
        FOR $b IN doc("bib.xml")//book[
            publisher=$p
        ]
        ORDERBY $b/price
        RETURN <book>
        </book>
    }</publisher>
}</publisher_list>
```

- **Query 10**: Find the maximum depth of the document named "partlist.xml"

```
NAMESPACE
xsd="http://ww.w3.org/2001/XMLSchemadatatypes"
FUNCTION depth(ELEMENT $e) RETURNS xsd:integer {
    -- An empty element has depth 1
    IF empty($e/*) THEN 1

    -- Otherwise, add 1 to max depth of children
    ELSE max(depth($e/*)) + 1
}

RETURN <result>{
    depth(doc("partlist.xml"))
}</result>
```

Simple Examples

Example 1

```
for $i in 1 to 3
return <oneEval>{$i}</oneEval>
```

<oneEval>1</oneEval>, <oneEval>2</oneEval>, <oneEval>3</oneEval>

```
let $i := (1 to 3)
return <oneEval>{$i}</oneEval>
```

<oneEval>1 2 3</oneEval>

Example 2

<oneEval>i is 1 and j is a</oneEval>,
<oneEval>i is 1 and j is b</oneEval>,
<oneEval>i is 2 and j is a</oneEval>,
<oneEval>i is 2 and j is b</oneEval>

XML File – catalog.xml

```
<catalog>
 <product dept="WMN">
  <number>557</number>
  <name language="en">Fleece Pullover</name>
  <colorChoices>navy black</colorChoices>
 </product>
 <product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
 </product>
 <product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
 </product>
 <product dept="MEN">
  <number>784</number>
  <name language="en">Cotton Dress Shirt</name>
  <colorChoices>white gray</colorChoices>
  <desc>Our <i>favorite</i> shirt!</desc>
 </product>
</catalog>
```

Example 2

```
(: double for-loop :)

for $i in (1, 2)
for $j in ("a", "b")
return <oneEval>i is {$i} and j is {$j}</oneEval>

for $i in (1, 2), $j in ("a", "b")
return <oneEval>i is {$i} and j is {$j}</oneEval>
```

Example 3

```
(: select products :)

for $prod in doc("catalog.xml")/catalog/product[@dept = 'ACC']
return $prod/name
```

<name language="en">Floppy Sun Hat</name>,
<name language="en">Deluxe Travel Bag</name>

```
for $prod in doc("catalog.xml")/catalog/product[@dept = 'ACC']
return $prod
```

Example 4

```
(: list of products in html :)

<html>
 <h1>Product Catalog</h1>
 <ul> {
    for $prod in doc("catalog.xml")/catalog/product
    return <li>number: {data($prod/number)},
          name: {data($prod/name)}</li>
 } </ul>
</html>
```

FLWORs

For – Let – Where – Order by – Return

Example 4

```
<html>
 <h1>Product Catalog</h1>
 <ul>
   <li>number: 557, name: Fleece Pullover</li>
   <li>number: 563, name: Floppy Sun Hat</li>
   <li>number: 443, name: Deluxe Travel Bag</li>
   <li>number: 784, name: Cotton Dress Shirt</li>
 </ul>
</html>
```

XML File – catalog.xml

```
<catalog>
 <product dept="WMN">
  <number>557</number>
  <name language="en">Fleece Pullover</name>
  <colorChoices>navy black</colorChoices>
 </product>
 <product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
 </product>
 <product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
 </product>
 <product dept="MEN">
  <number>784</number>
  <name language="en">Cotton Dress Shirt</name>
  <colorChoices>white gray</colorChoices>
  <desc>Our <i>favorite</i> shirt!</desc>
 </product>
</catalog>
```

Example 5

```
(:  count number of entries :)

<html>
 <h1>Product Catalog</h1>
 <p>A <i>huge</i> list of {count(doc("catalog.xml")//product)}
        products.</p>
</html>
```

Example 6

```
(: using the where clause :)

for $prod in doc("catalog.xml")//product
let $prodDept := $prod/@dept
where $prodDept = "ACC" or $prodDept = "WM"
return $prod/name


<name language="en">Fleece Pullover</name>,
<name language="en">Floppy Sun Hat</name>,
<name language="en">Deluxe Travel Bag</name>
```

Example 7

```
(: same example, basically :)
(: intermingled for and let clauses :)

let $doc := doc("catalog.xml")
for $prod in $doc//product
let $prodDept := $prod/@dept
let $prodName := $prod/name
where $prodDept = "ACC" or $prodDept = "WMN“
return $prodName
```

Example 8

```
for $prod in doc("catalog.xml")//product
let $d := $prod/@dept
let $n := data($prod/number)
return <result dept="{$d}" number="{$n}"/>
```

Example 8

```
(: what does this do? :)
let $prods := doc("catalog.xml")//product
for $d in distinct-values($prods/@dept),
  $n in distinct-values($prods[@dept = $d]/number)
return <result dept="{$d}" number="{$n}"/>
```

How many results?
What if we just removed the first or second “distinct-values”?
Could we have avoided using function distinct-values?

XML File – catalog.xml

```
<catalog>
 <product dept="WMN">
  <number>557</number>
  <name language="en">Fleece Pullover</name>
  <colorChoices>navy black</colorChoices>
 </product>
 <product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
 </product>
 <product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
 </product>
 <product dept="MEN">
  <number>784</number>
  <name language="en">Cotton Dress Shirt</name>
  <colorChoices>white gray</colorChoices>
  <desc>Our <i>favorite</i> shirt!</desc>
 </product>
</catalog>
```

Example 8

```
<result dept="WMN" number="557"/>,
 <result dept="ACC" number="563"/>,
 <result dept="ACC" number="443"/>,
 <result dept="MEN" number="784"/>
```

XML File – order.xml

```
<order num="00299432" date="2006-09-15" cust="0221A">
 <item dept="WMN" num="557" quantity="1" color="navy"/>
 <item dept="ACC" num="563" quantity="1"/>
 <item dept="ACC" num="443" quantity="2"/>
 <item dept="MEN" num="784" quantity="1" color="white"/>
 <item dept="MEN" num="784" quantity="1" color="gray"/>
 <item dept="WMN" num="557" quantity="1" color="black"/>
</order>
```

Example 9

(: join :)

```
for $item in doc("order.xml")//item,
        $product in doc("catalog.xml")//product
where $item/@num = $product/number
return <item num="{$item/@num}"
        name="{$product/name}"
        quan="{$item/@quantity}"/>
```

Can we re-write the query without the where clause?

Example 11

(: order! :)

```
for $item in doc("order.xml")//item
order by $item/@num
return $item

for $item in doc("order.xml")//item
order by $item/@dept, $item/@num
return $item
```

Example 9

```
<item num="557" name="Fleece Pullover" quan="1"/>,
<item num="563" name="Floppy Sun Hat" quan="1"/>,
<item num="443" name="Deluxe Travel Bag" quan="2"/>,
<item num="784" name="Cotton Dress Shirt" quan="1"/>,
<item num="784" name="Cotton Dress Shirt" quan="1"/>,
<item num="557" name="Fleece Pullover" quan="1"/>
```

Example 11

```
<item dept="ACC" num="443" quantity="2"/>,
<item dept="WMN" num="557" quantity="1" color="black"/>,
<item dept="WMN" num="557" quantity="1" color="navy"/>,
<item dept="ACC" num="563" quantity="1"/>,
<item dept="MEN" num="784" quantity="1" color="white"/>,
<item dept="MEN" num="784" quantity="1" color="gray"/>
```

Example 10

(: same join, no where clause :)

```
for $item in doc("order.xml")//item,
    $product in doc("catalog.xml")//product[number = $item/@num]
return <item num="{$item/@num}"
                name="{$product/name}“
                quan="{$item/@quantity}"/>
```

Example 11

```
<item dept="ACC" num="443" quantity="2"/>,
<item dept="ACC" num="563" quantity="1"/>,
<item dept="MEN" num="784" quantity="1" color="white"/>,
<item dept="MEN" num="784" quantity="1" color="gray"/>,
<item dept="WMN" num="557" quantity="1" color="black"/>,
<item dept="WMN" num="557" quantity="1" color="navy"/>
```

Example 12

```
(: what does this do? :)
for $d in distinct-values(doc("order.xml")//item/@dept)
let $items := doc("order.xml")//item[@dept = $d]
order by $d
return <department code="{$d}">{
        for $i in $items
        order by $i/@num
        return $i
        }</department>
```

Example 13

```
<department code="ACC" numItems="2" distinctItemNums="2" totQuant="3"/>,
<department code="MEN" numItems="2" distinctItemNums="1" totQuant="2"/>,
<department code="WMN" numItems="2" distinctItemNums="1" totQuant="2"/>
```

Example 12

```
<department code="ACC">
  <item dept="ACC" num="443"  quantity="2"/>
  <item dept="ACC" num="563" quantity="1"/>
</department>,
<department code="MEN">
  <item dept="MEN" num="784" quantity="1" color="gray"/>
  <item dept="MEN" num="784" quantity="1" color="white"/>
</department>,
<department code="WMN">
  <item dept="WMN" num="557" quantity="1" color="black"/>
  <item dept="WMN" num="557" quantity="1" color="navy"/>
</department>
```

Extras

Example 13

```
(: aggregate by group :)

for $d in distinct-values(doc("order.xml")//item/@dept)
let $items := doc("order.xml")//item[@dept = $d]
order by $d
return <department code="{$d}"
        numItems="{count($items)}"
        distinctItemNums="{count(distinct-values($items/@num))}"
        totQuant="{sum($items/@quantity)}"/>
```

Enclosed expressions that evaluate to attributes

```
for $prod in doc("catalog.xml")/catalog/product
return <li>{$prod/@dept}number: {$prod/number}</li>

<li dept="WMN">number: <number>557</number></li>,
<li dept="ACC">number: <number>563</number></li>,
<li dept="ACC">number: <number>443</number></li>,
<li dept="MEN">number: <number>784</number></li>
```

Multiple conditions in where clause

```
for $prod in doc("catalog.xml")//product
let $prodDept := $prod/@dept
where $prod/number > 100
        and starts-with($prod/name, "F")
        and exists($prod/colorChoices)
        and ($prodDept = "ACC" or $prodDept = "WMN")
return $prod
```

Three-way join in a where clause

```
for $item in doc("order.xml")//item,
        $product in doc("catalog.xml")//product,
        $price in doc("prices.xml")//prices/priceList/prod
where $item/@num = $product/number
        and $product/number = $price/@num
return <item num="{$item/@num}"
        name="{$product/name}"
        price="{$price/price}"/>
```

Using an empty order declaration

```
declare default order empty greatest;
for $item in doc("order.xml")//item
order by $item/@color
return $item
```

```
<item dept="WMN" num="557" quantity="1" color="black"/>,
<item dept="MEN" num="784" quantity="1" color="gray"/>,
<item dept="WMN" num="557" quantity="1" color="navy"/>,
<item dept="MEN" num="784" quantity="1" color="white"/>,
<item dept="ACC" num="443" quantity="2"/>,
<item dept="ACC" num="563" quantity="1"/>
```

XML File – prices.xml

```
<prices>
 <priceList effDate="2006-11-15">
  <prod num="557">
   <price currency="USD">29.99</price>
   <discount type="CLR">10.00</discount>
  </prod>
  <prod num="563">
   <price currency="USD">69.99</price>
  </prod>
  <prod num="443">
   <price currency="USD">39.99</price>
   <discount type="CLR">3.99</discount>
  </prod>
 </priceList>
</prices>
```