Query Languages for XML

csc343, Introduction to Databases
Nosayba El-Sayed

Slides from Diane Horton, with material from Ryan Johnson, Manos Papagelis, Jeff Ullman, Ramona Truta, and Renée Miller
Fall 2015

UNIVERSITY OF TORONTO

DCS50

Previously, on CSC343...

- XML is great for
  - Recording data that software needs.
  - Exchange of information between pieces of software.
- XML is said to be "self-describing".
  - Schema-like information is part of the data itself.
  - Example:

```
<student stnum="1234" name="Cindylou Who">
  <address>
    <street>99 Alfalfa Way</street>
    <city>Whoville</city>
  </address>
</student>
```

Previously, on CSC343…

- Well-formed XML
  - Just need a single *root* element and proper *nesting* (all elements must have a closing tag).
  - Any tag or attribute can go anywhere.
- Valid XML
  - A valid XML must be well-formed + conforms to a DTD
  - A "DTD" (document type definition) specifies what tags and attributes are permitted, where they can go, and how many there must be.
  - A valid XML file is one that has a DTD and follows the rules specified in its DTD.

Quiz.xml (Example)

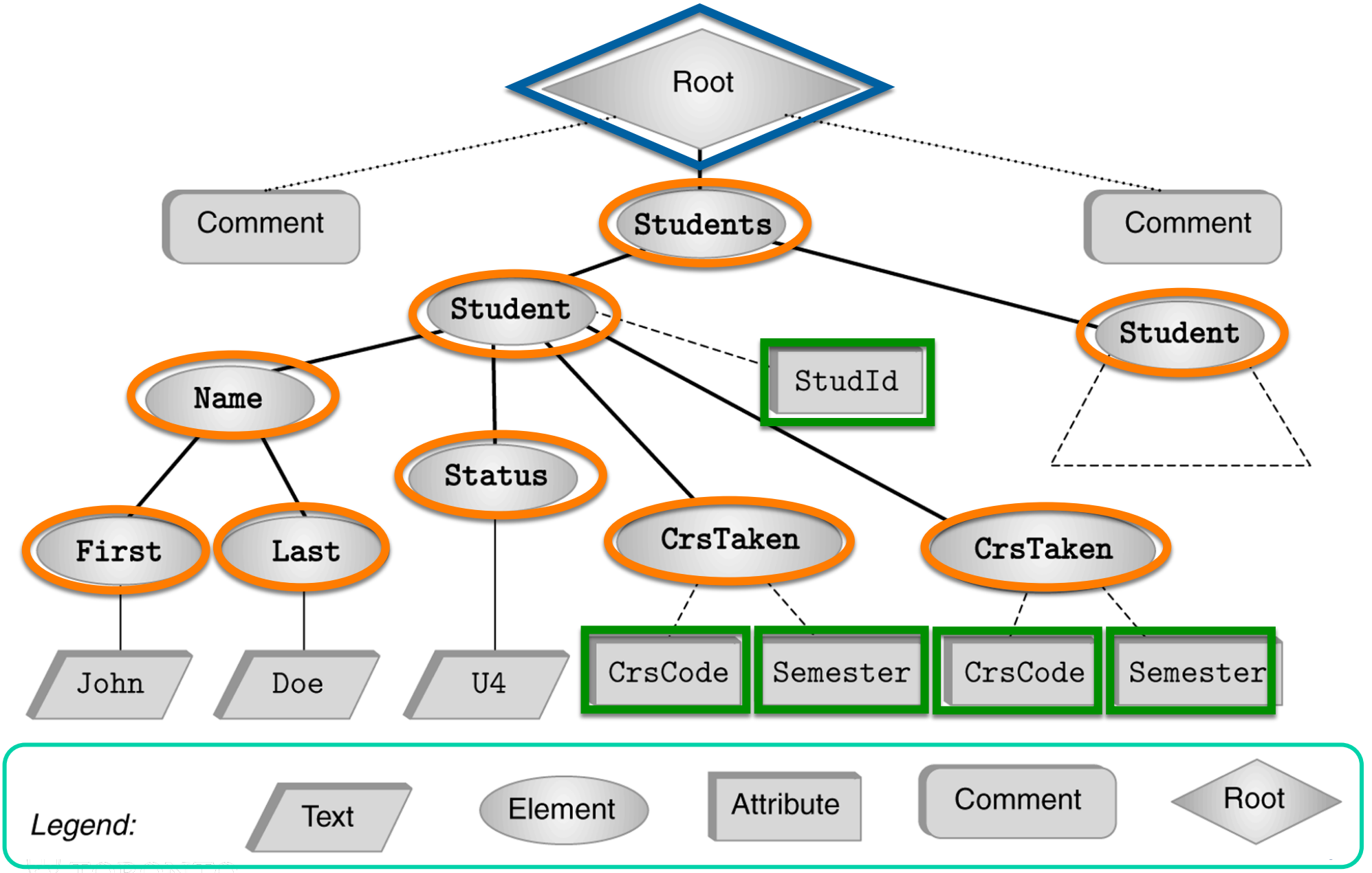
```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Quiz SYSTEM "quiz.dtd">
<Quiz quizID="csc343" title="Homework Set 1”>
  <Question QID="N-15" weight="2"/>
  <Question QID="TF-01" weight="1"/>
  <Question QID="MC-05” weight="3"/>
  <Question QID="MC-08” weight="2"/>
</Quiz>
```

Quiz.dtd Example

```
<!ELEMENT Quiz (Question+)>
<!ATTLIST Quiz quizID CDATA #REQUIRED>
<!ATTLIST Quiz title CDATA #REQUIRED>
<!ATTLIST Quiz hints (yes|no) #REQUIRED>
<!ELEMENT Question EMPTY>
<!ATTLIST Question QID ID #REQUIRED>
<!ATTLIST Question weight CDATA #REQUIRED>
```

XML DATA MODEL

XML Document is a Tree



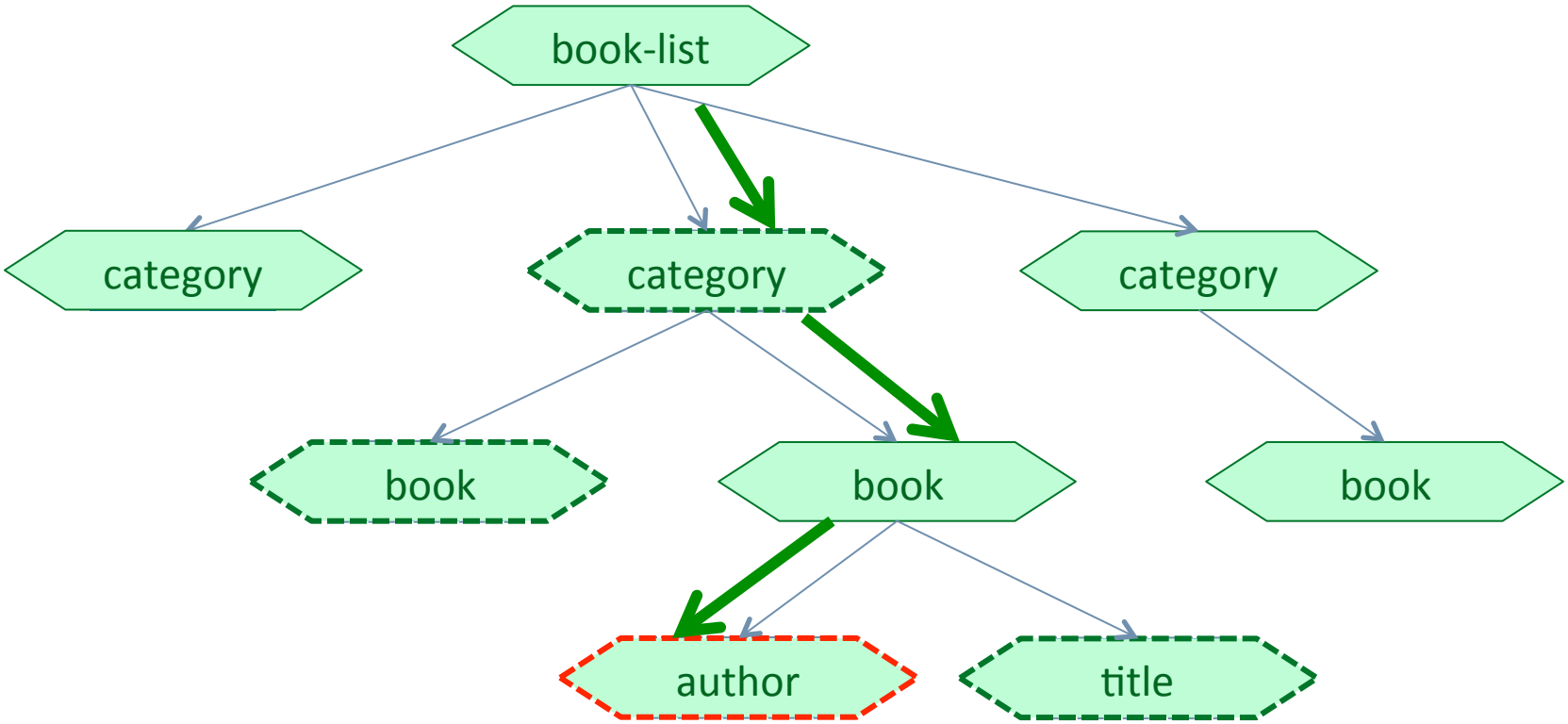
Manipulating XML

- Once we have the data we want to...
  - extract parts of interest
  - transform document
  - relate (= join) different parts of a file (or different files)
- In other words, we need queries!
- XML Query Languages
  - XPATH
  - XQUERY

XPath Query Language

Queries over hierarchical data

- Observe: all XML documents are trees
  - Each element has one "path" to the document root
  - Similar to a file system



Example

```
<?xml  version="1.0" ?>
<Students>
  <Student StudId="111111111" >
     <Name><First>John</First><Last>Doe</Last></Name>
     <Status>U2</Status>
     <CrsTaken CrsCode="CS308" Semester="F1997" />
     <CrsTaken CrsCode="MAT123" Semester="F1997" />
  </Student>
  <Student StudId="987654321" >
     <Name><First>Bart</First><Last>Simpson</Last></Name>
     <Status>U4</Status>
     <CrsTaken CrsCode="CS308" Semester="F1994" />
  </Student>
</Students>
```

- To find all course codes, we use this path:
  Students → Student → CrsTaken → CrsCode attribute

Queries over hierarchical data

- Slash "/" usage, similar to file systems
- Specify full or partial paths..
  - /Students/Student/Name

  ⇒ Returns the Name of each Student element under Students

- ...with selection:
  - /Students/Student[Status='U2']

  => Returns any Student whose Status is 'U2'

Writing and Running an XPath query

- Create a file containing:
  `fn:doc(“«xml file»”)«path expression»`
  - `fn:doc` is a function that parses the document and evaluates to a document tree.
- Suppose `query.xq` contains:
  `fn:doc(“courses.xml”)/Student/CrsTaken/@CrsCode`
  - Each slash takes us down one level in the tree.
  - `@` takes us to an *attribute*; otherwise we go to a sub-element.
- To run it on cdf:
  `galax-run query.xq`

Result of a path expression

- The result of a path expression is a sequence of items from the document.
- Each item is either
  - a primitive value, such as a string or integer
  - or a node in the document.

Homogeneous or heterogeneous results

- Often, queries yield homogeneous results.
  Examples:
  ```
  doc("quiz.xml")//questions/mc-question
  doc("quiz.xml")//tf-question/@solution
  ```

- But some queries don't.
  Example:
  ```
  doc("quiz.xml")/quiz/questions/*/*
  ```
  Yields a mix of question elements and option elements.

Demo

- Extract things from quiz.xml

1. Find the solution to every question (regardless of type)
2. Get the elements of the true-false questions only
3. Get the text of MC question that has QID “Q8888”
4. Get *second* response for student with sid s555555555.

XPath documentation

- Official Xpath documentation:
  http://www.w3.org/TR/xpath20/
- Functions and operators (very useful!):
  http://www.w3.org/TR/xpath-functions/
- Manual (available on cdf):
  /usr/share/doc/galax-doc/manual/manual.html
  (Relevant if installing galax on your own machine.)

Other axes

Axes

- So far, we've navigated the tree by going from parent to child node.
- There are many more modes of navigation, called axes.
- Here, axes is the plural of axis, not axe!

Syntax for axes

- Notation:
  /«axis»::
  where *axis* is one of
  - `child`
  - `parent`
  - `attribute` (we'll see more axes later)
- If you do not specify an axis, the default is used: `child`
- So the path expression

  ```
  fn:doc("courses.xml")/Students
  ```

  is shorthand for

  ```
  fn:doc("courses.xml")/child::Students
  ```

@ is shorthand for the attribute axis

- So this path expression

```
fn:doc("courses.xml")
    /Students
    /Student
    /CrsTaken
    /@CrsCode
```

  is short for

```
fn:doc("courses.xml")
    /child::Students
    /child::Student
    /child::CrsTaken
    /attribute::CrsCode
```

Attribute axis in a condition

- This path expression

```
fn:doc("courses.xml")
    /Students
    /Student
    /CrsTaken[@CrsCode="cs308"]
```

  is short for

```
fn:doc("courses.xml")
    /child::Students
    /child::Student
    /child::CrsTaken[attribute::CrsCode="cs308"]
```

Other shorthand for axes

- // is shorthand for the descendant-or-self axis, so this

```
fn:doc("courses.xml")
    //CrsTaken
```

is short for

```
fn:doc("courses.xml")
    /descendant-or-self::CrsTaken
```

- Dot (.) is shorthand for the self axis, so this

```
fn:doc("courses.xml")
    //CrsTaken@CrsCode[.="cs308"]
```

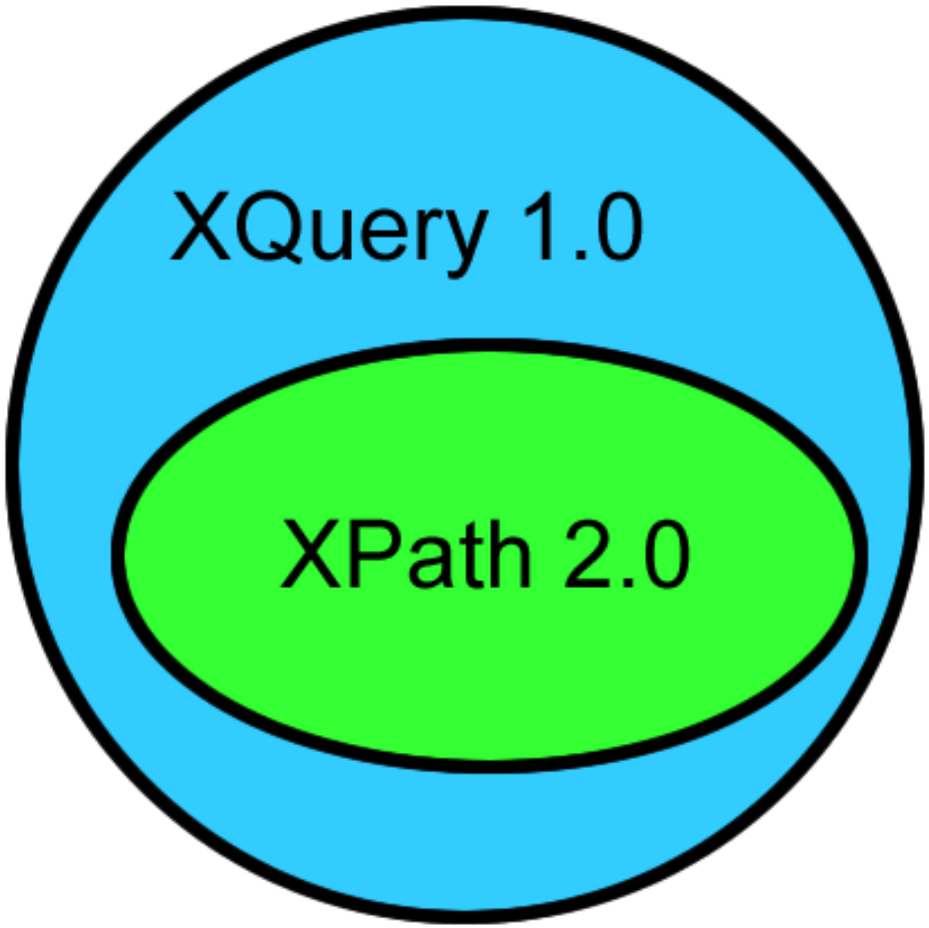
is short for

```
fn:doc("courses.xml")
    /descendant-or-self::CrsTaken
    /attribute::CrsCode[self::="cs308"]
```

And there are even more axes

- Other axes include:
  - `parent`
  - `ancestor`
  - `ancestor-or-self`
  - `following-sibling`
  - `preceding-sibling`
- See section 2.2 of the documentation for more: `http://www.w3.org/TR/xpath/#axes`

XQuery Query Language



Intro

- XQuery extends XPath.
- It uses the same data model.
  - A document is a tree.
  - A query result is a sequence of items from the document.

- XQuery is an expression language.
  - Any XQuery expression can be an argument of any other XQuery expression.

Item sequences are flattened

- XQuery sometimes generates nested sequences.
- These are always flattened.
- Example:

```
(1 2 () (3 4)) = (1 2 3 4)
```

FLWOR expressions

- Example:

```
let $d := fn:doc("bank.xml")
for $tfq in $d//TFQuestion
where $tfq/@answer="True"
order by $qid
return $tfq/question
```

- The semantics of *return* is surprising:
  - It does not terminate the FLWOR expression!
  - It specifies the value produced by the current iteration.
  - The sequence of these is the result of the FLWOR expression.

Notes about the syntax

- Keywords are case-sensitive.
- Variables begin with **$**.
- Rule: (for | let)**+** where**?** order-by**?** **return**
- Remember that XQuery is an expression language.
  - A FLWOR expression has subexpressions.

```
let $d := fn:doc("bank.xml")
for $tfq in $d//TFQuestion
where $tfq/@answer="True"
order by $qid
return $tfq/question
```

  - Any of these could itself be a FLWOR expression or other complex expression.

For vs let

- For is like

```
for x in [99, 42, 101, 5]
```

  - It iterates over the items in a sequence.
  - Each time, the variable gets a new value.

- Let is like

```
x = [99, 42, 101, 5]
```

  - No iteration occurs.
  - x gets one value, which is a sequence.

Order-by

- Form: `order by` *«expression»*
- We can optionally specify `ascending` or `descending`.
- The expression is evaluated for each assignment to variables.
- Its value determines placement of the FLWOR expression's result in the output sequence.

Mixing static output and evaluated expressions

- Lets us construct new XML structures with our code.
- Example:

```
<title>Facts about Canada</title>
<truth>
{
  let $d := fn:doc("bank.xml")
  return $d//tf-question[@solution="true"]/question
}
</truth>
<lies>
{
  let $d := fn:doc("bank.xml")
  return $d//tf-question[@solution="false"]/question
}
</lies>
```

What's evaluated and what's not?

- The default: don't evaluate.
  - Example:
    `<title>$x</title>`
  - This evaluates to a title element with value "`$x`"
- To override the default and force evaluation, surround with braces.
  - Example:
    `<title>{$x}</title>`

Generous comparison

- If A and B are sequences, A=B means
  ∃x∈A, y∈B such that x=y.
- Examples:
  - `(1,2) = (2,3)` is true.
  - This path expression:
    `fn:doc("races.xml")//race[result < 3.50]`
    yields races that include *any* result less than 3.50.

Strict comparison

- Alternative: The comparison operators

  `eq  ne  lt  le  gt  ge`

  succeed only if both sequences have length one.
- Example:

```
fn:doc("races.xml")
                //race[sponsor eq "HarryRosen"]
```

  is true if the LHS yields a sequence of length one that is `"HarryRosen"`.

Eliminating duplicates

- Apply function `distinct-values` to a sequence.
- Subtlety:
  - It strips tags away from elements and compares the string values.
  - But it doesn't restore the tags in the result.
- Example:

```
let $d := fn:doc("races.xml")
return distinct-values($d//result)
```

More kinds of expressions

Branching expressions

- Form: `if («E1») then «E2» else «E3»`
- All three parts are required.
- Value of the if expression is
  - *E2* if the EBV of *E1* is true, and
  - *E3* if the EBV of *E1* is false.
    (EBV = Effective Boolean Value)
- Example:

```
if ($q/@solution="True")
then $q/question else ()
```

Any type can be treated as boolean

- Like many languages, we can treat anything as boolean.
- The effective boolean value (EBV) of an expression is:
  - the value of the expression, if it is already of type boolean
  - otherwise it is
    - FALSE if the expression evaluates to 0, "", or ().
    - TRUE if not.
- Example:

```
let $d := fn:doc("races.xml")
return
  if ($d//result[@who="r1"])
  then <yay/>
  else <nay/>
```

Quantifier expressions

- Form: `some` *«variable»* `in` *«E1»* `satisfies` *«E2»*
- Meaning
  - Evaluate *E1*, yielding a sequence.
  - Let the variable be each item in the sequence, and evaluate *E2* for each.
  - The value of the whole expression is true if *E2* has EBV true at least once.
- Form: `every` *«variable»* `in` *«E1»* `satisfies` *«E2»*
- Meaning is analogous.

Comparisons based on document order

- Form: *«E1»* << *«E2»* and *«E1»* >> *«E2»*
- Meaning: comes before (or after) in the document.
- Example:

```
let $d := fn:doc("races.xml")
return
   $d//race[@name="WaterfrontMarathon"]
   <<
   $d//race[@name="HarryRosen"]
```

Output, given our "races.xml" file?