# Query Languages for XML

XPath

XQuery

XSLT

# The XPath/XQuery Data Model

- Corresponding to the fundamental "relation" of the relational model is: *sequence of items*.
- An *item* is either:
  1. A primitive value, e.g., integer or string.
  2. A *node* (defined next).

# Principal Kinds of Nodes

1. *Document nodes* represent entire documents.
2. *Elements* are pieces of a document consisting of some opening tag, its matching closing tag (if any), and everything in between.
3. *Attributes* names that are given values inside opening tags.

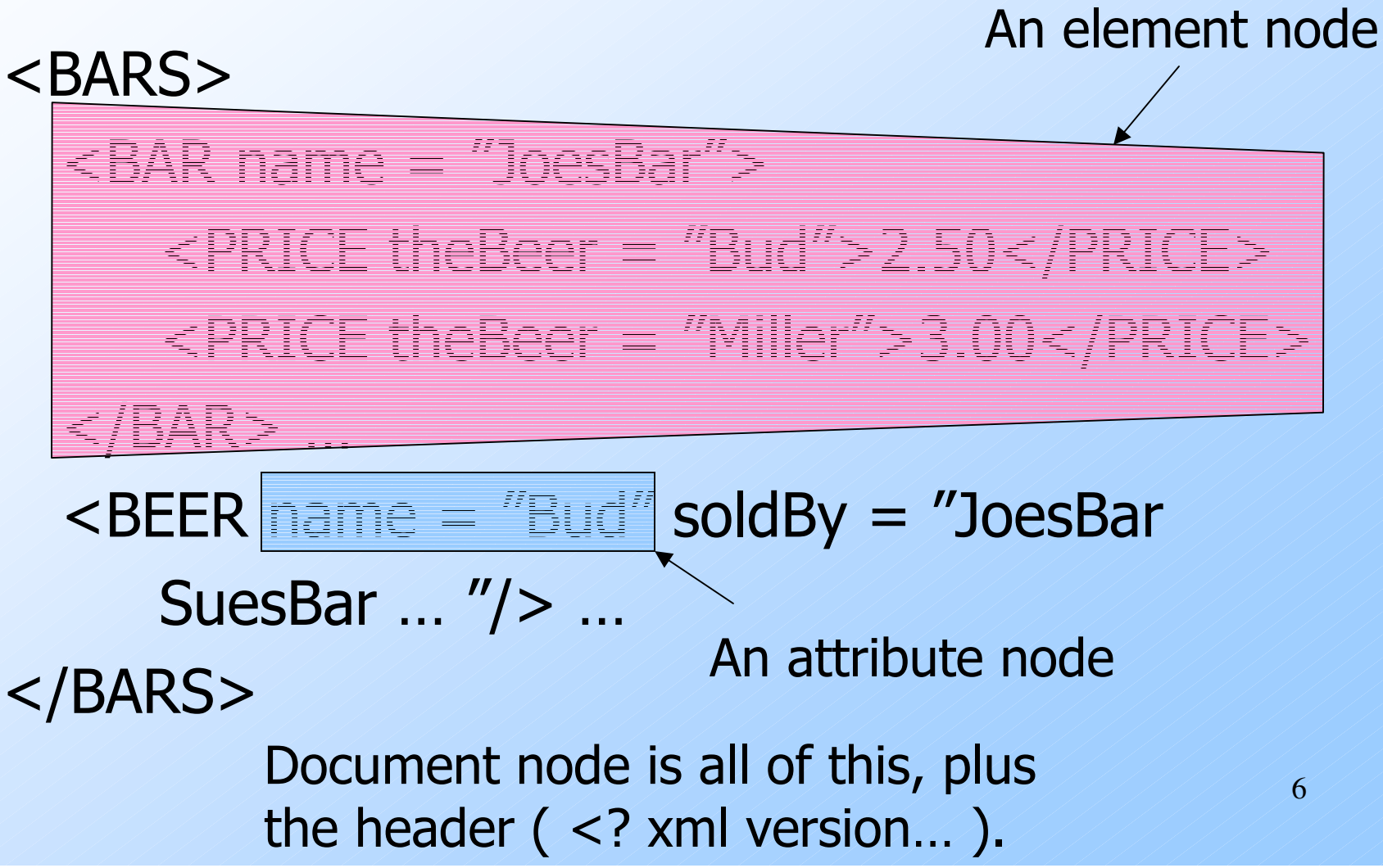
# Document Nodes

- Formed by doc(URL) or document(URL).
- Example: doc(/usr/class/cs145/bars.xml)
- All XPath (and XQuery) queries refer to a doc node, either explicitly or implicitly.
  - Example: key definitions in XML Schema have Xpath expressions that refer to the document described by the schema.

# DTD for Running Example

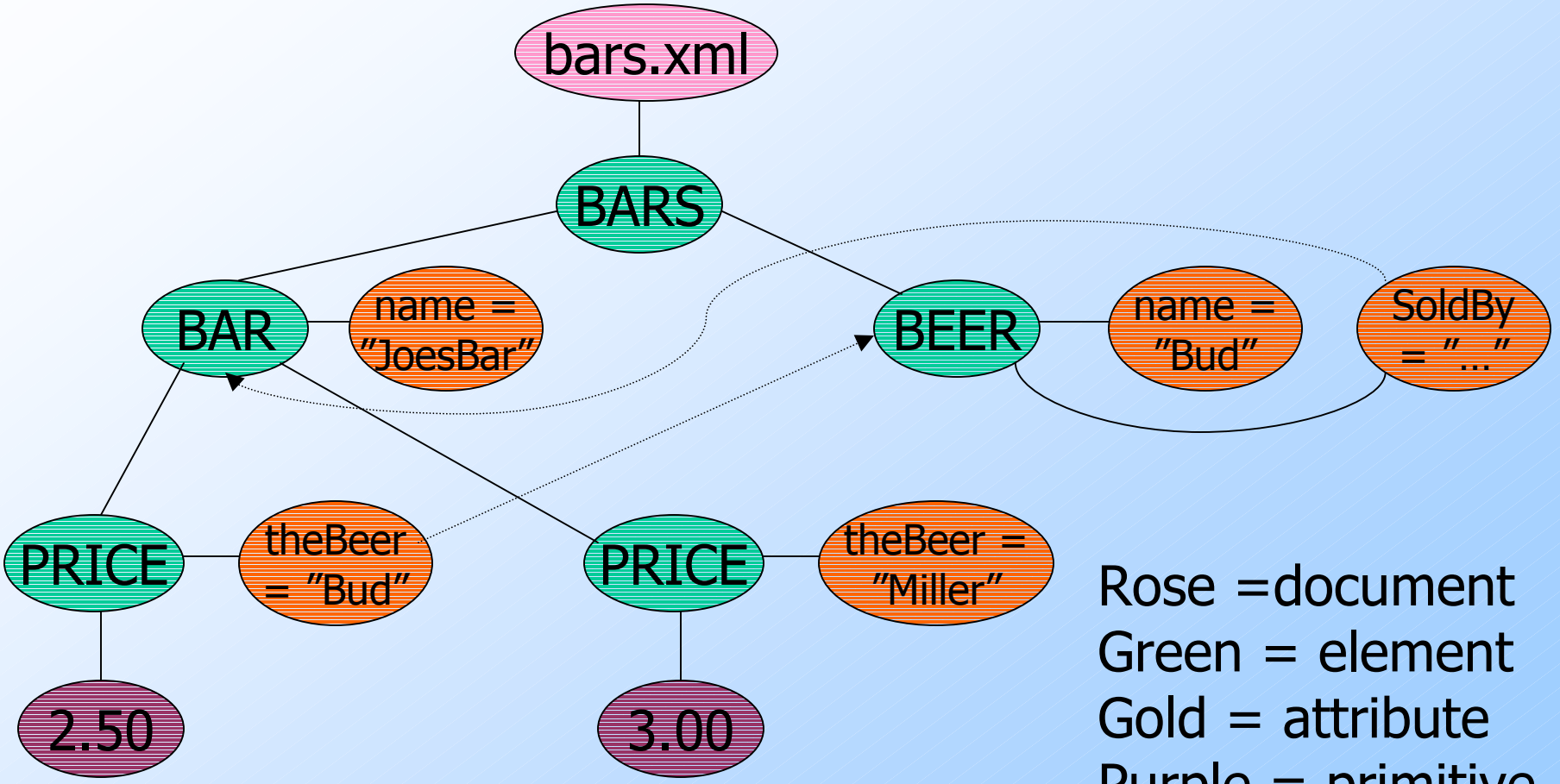
```
<!DOCTYPE BARS [
   <!ELEMENT BARS (BAR*, BEER*)>
   <!ELEMENT BAR (PRICE+)>
        <!ATTLIST BAR name ID #REQUIRED>
   <!ELEMENT PRICE (#PCDATA)>
        <!ATTLIST PRICE theBeer IDREF #REQUIRED>
   <!ELEMENT BEER EMPTY>
        <!ATTLIST BEER name ID #REQUIRED>
        <!ATTLIST BEER soldBy IDREFS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

# Example Document




Document node is all of this, plus the header ( <? xml version... ).

# Nodes as Semistructured Data




Rose =document
Green = element
Gold = attribute
Purple = primitive value

# Paths in XML Documents

- XPath is a language for describing paths in XML documents.
- The result of the described path is a sequence of items.

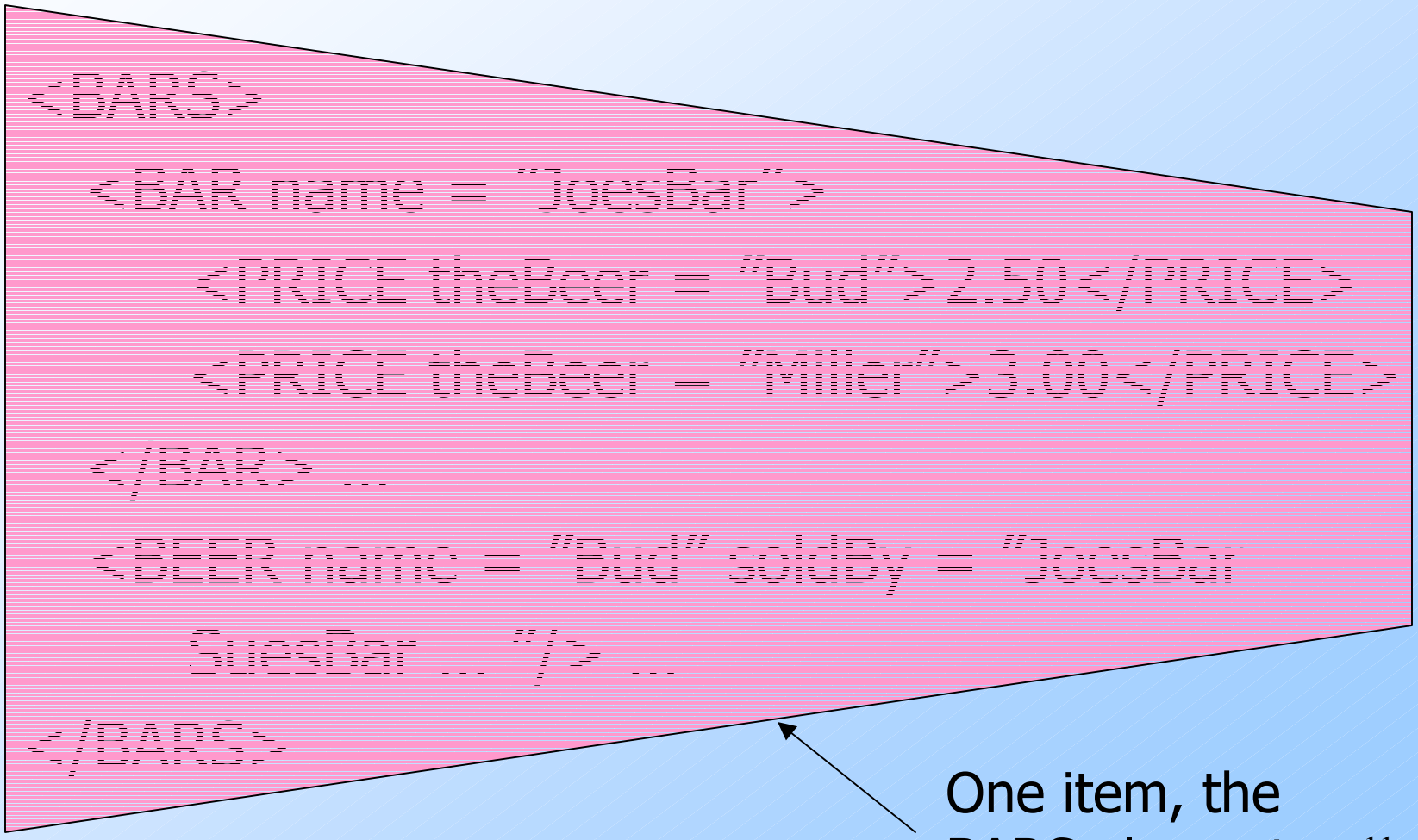
# Path Expressions

- Simple path expressions are sequences of slashes (/) and tags, starting with /.
  - Example: /BARS/BAR/PRICE
- Construct the result by starting with just the doc node and processing each tag from the left.

# Evaluating a Path Expression

- Assume the first tag is the root.
  - Processing the doc node by this tag results in a sequence consisting of only the root element.
- Suppose we have a sequence of items, and the next tag is *X*.
  - For each item that is an element node, replace the element by the subelements with tag *X*.

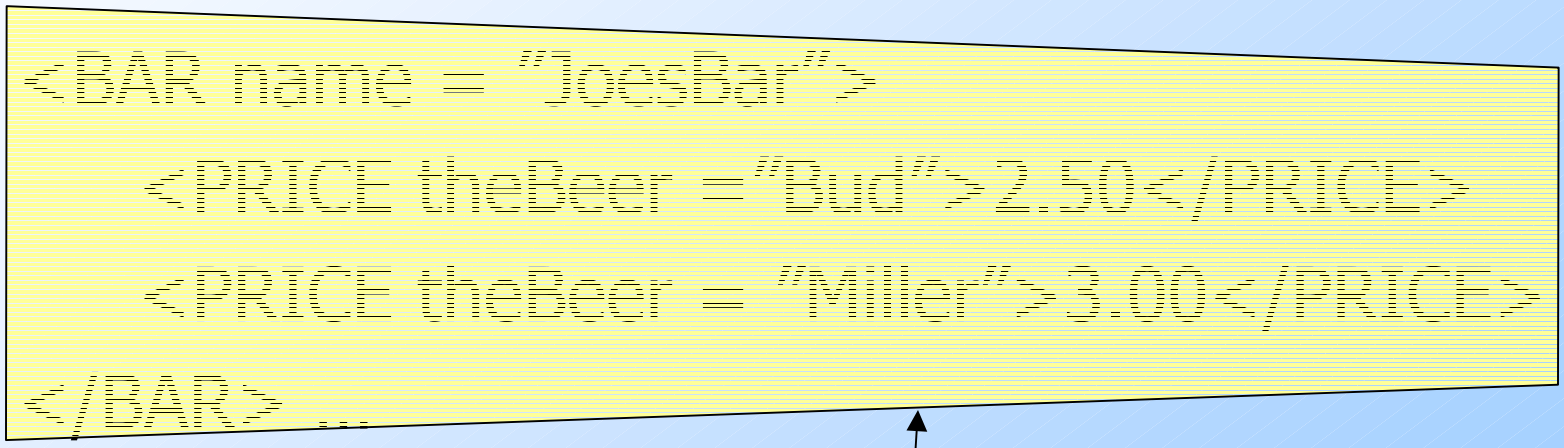
# Example: /BARS




One item, the BARS element

# Example: /BARS/BAR

<BARS>




<BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/> ...

</BARS>

This BAR element followed by all the other BAR elements

# Example: /BARS/BAR/PRICE

<BARS>

<BAR name = ”JoesBar”>

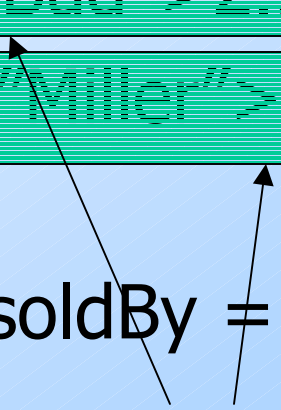
<PRICE theBeer =”Bud”>2.50</PRICE>

<PRICE theBeer = ”Miller”>3.00</PRICE>

</BAR> …

<BEER name = ”Bud” soldBy = ”JoesBar SuesBar …”/> …

</BARS>



These PRICE elements followed by the PRICE elements of all the other bars.

# Attributes in Paths

- Instead of going to subelements with a given tag, you can go to an attribute of the elements you already have.
- An attribute is indicated by putting @ in front of its name.

# Example: /BARS/BAR/PRICE/@theBeer

```
<BARS>
  <BAR name = ”JoesBar”>
      <PRICE theBeer = ”Bud”>2.50</PRICE>
      <PRICE theBeer = ”Miller”>3.00</PRICE>
  </BAR> …
  <BEER name = ”Bud” soldBy = ”JoesBar
      SuesBar …”/> …
</BARS>
```

These attributes contribute ”Bud” ”Miller” to the result, followed by other theBeer values.

# Remember: Item Sequences

- Until now, all item sequences have been sequences of elements.
- When a path expression ends in an attribute, the result is typically a sequence of values of primitive type, such as strings in the previous example.

# Paths that Begin Anywhere

- If the path starts from the document node and begins with //*X*, then the first step can begin at the root or any subelement of the root, as long as the tag is *X*.

# Example: //PRICE

<BARS>

<BAR name = ”JoesBar”>

<PRICE theBeer =”Bud”>2.50</PRICE>

<PRICE theBeer = ”Miller”>3.00</PRICE>

</BAR> …

<BEER name = ”Bud” soldBy = ”JoesBar SuesBar …”/> …

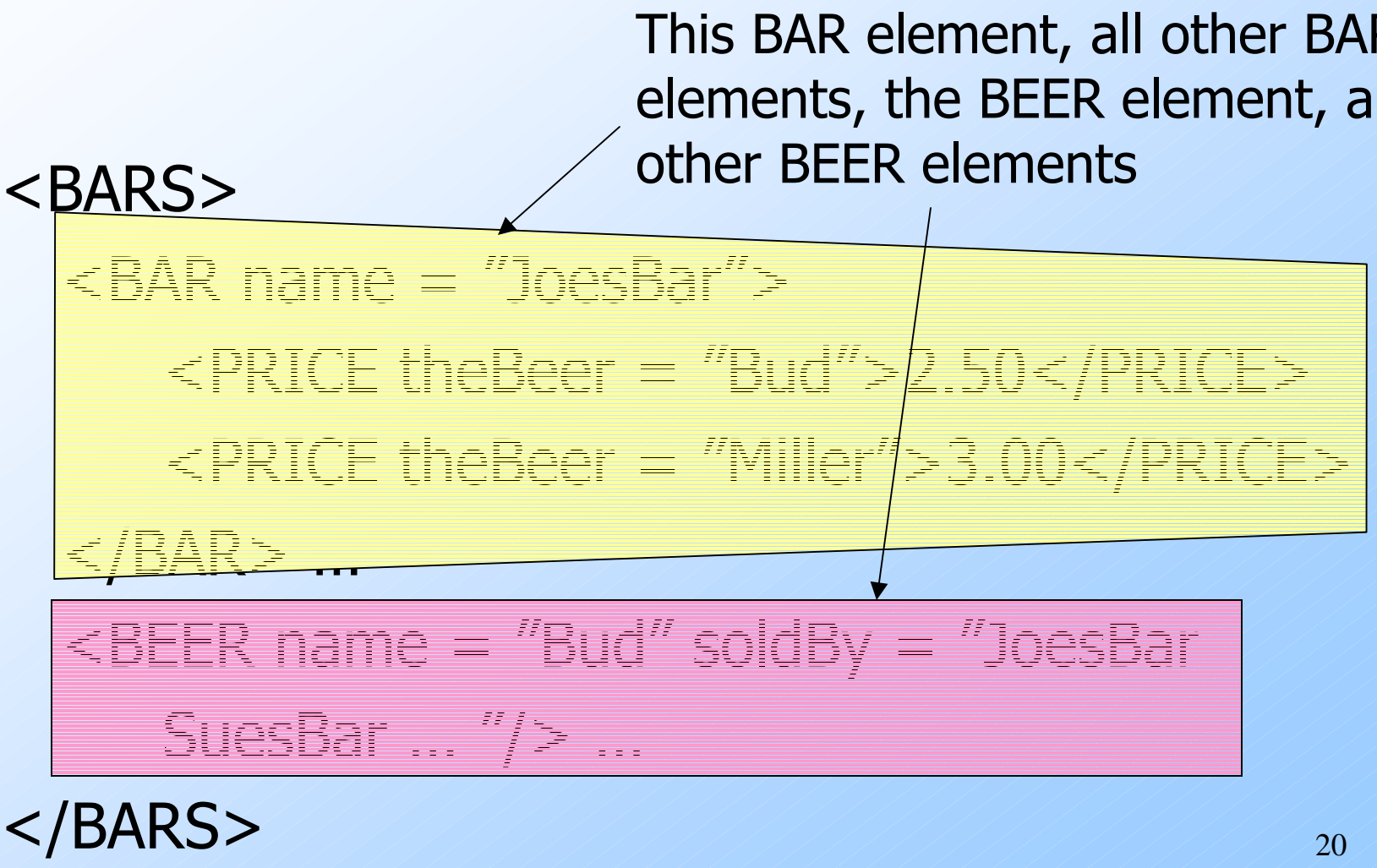
</BARS>

These PRICE elements and any other PRICE elements in the entire document

# Wild-Card *

- A star (*) in place of a tag represents any one tag.
- Example: /*/*/PRICE represents all price objects at the third level of nesting.

# Example: /BARS/*

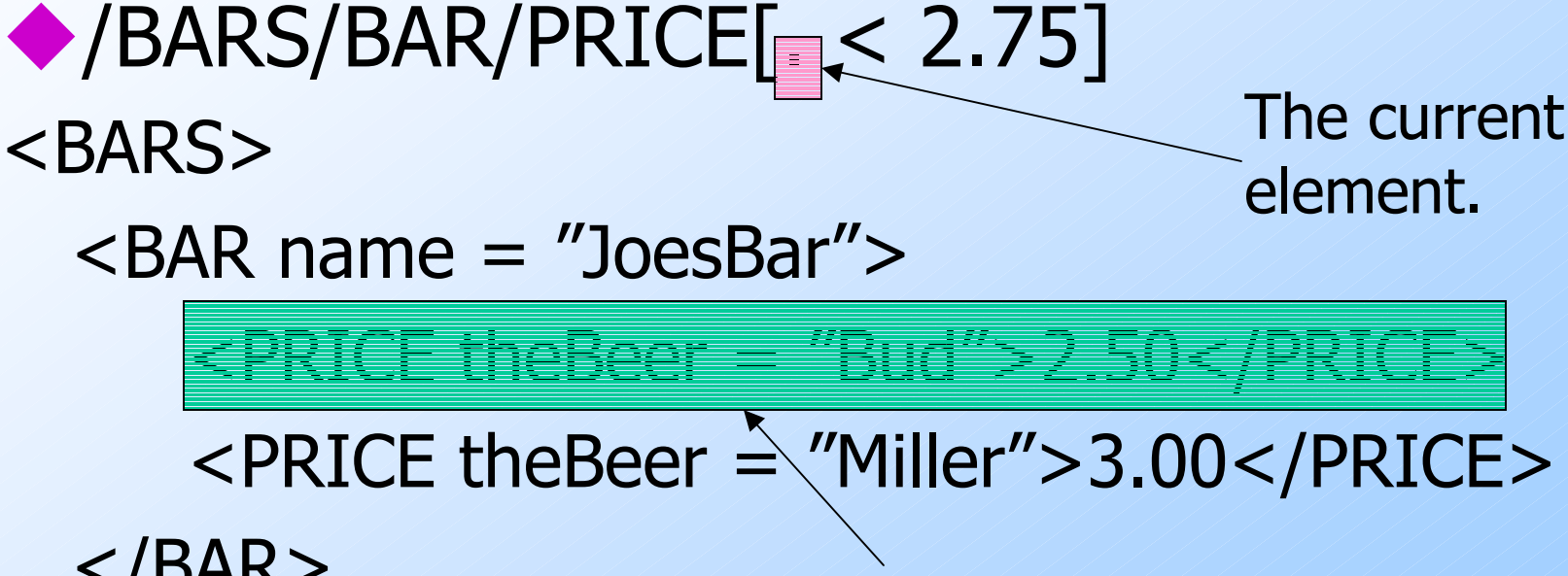


# Selection Conditions

- A condition inside [...] may follow a tag.
- If so, then only paths that have that tag and also satisfy the condition are included in the result of a path expression.

# Example: Selection Condition

- /BARS/BAR/PRICE[. < 2.75]



# Example: Attribute in Selection

- /BARS/BAR/PRICE[@theBeer = ”Miller”]

```
<BARS>
  <BAR name = ”JoesBar”>
      <PRICE theBeer = ”Bud”>2.50</PRICE>
      <PRICE theBeer = ”Miller”>3.00</PRICE>
  </BAR> ...
```

Now, this PRICE element is selected, along with any other prices for Miller.

# Axes

- In general, path expressions allow us to start at the root and execute steps to find a sequence of nodes at each step.
- At each step, we may follow any one of several *axes*.
- The default axis is child:: --- go to all the children of the current set of nodes.

# Example: Axes

- /BARS/BEER is really shorthand for /BARS/child::BEER .
- @ is really shorthand for the attribute:: axis.
  - Thus, /BARS/BEER[@name = ”Bud” ] is shorthand for

    /BARS/BEER[attribute::name = ”Bud”]

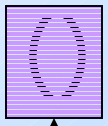
# More Axes

- Some other useful axes are:
  1. parent:: = parent(s) of the current node(s).
  2. descendant-or-self:: = the current node(s) and all descendants.
     - Note: // is really shorthand for this axis.
  3. ancestor::, ancestor-or-self, etc.
  4. self (the dot).

# XQuery

- XQuery extends XPath to a query language that has power similar to SQL.
- Uses the same sequence-of-items data model.
- XQuery is an expression language.
  - Like relational algebra --- any XQuery expression can be an argument of any other XQuery expression.

# More About Item Sequences

- XQuery will sometimes form sequences of sequences.
- All sequences are flattened.
- Example: (1 2  (3 4)) = (1 2 3 4).

Empty sequence

# FLWR Expressions

1. One or more for and/or let clauses.
2. Then an optional where clause.
3. A return clause.

# Semantics of FLWR Expressions

- Each for creates a loop.
  - let produces only a local definition.
- At each iteration of the nested loops, if any, evaluate the where clause.
- If the where clause returns TRUE, invoke the return clause, and append its value to the output.

# FOR Clauses

for <variable> in <expression>, . . .

- Variables begin with $.
- A for-variable takes on each item in the sequence denoted by the expression, in turn.
- Whatever follows this for is executed once for each value of the variable.

# Example: FOR

Our example BARS document

"Expand the enclosed string by replacing variables and path exps. by their values."

```
for $beer in
   document("bars.xml")/BARS/BEER/@name
return
   <BEERNAME>{$beer}</BEERNAME>
```

- $beer ranges over the name attributes of all beers in our example document.
- Result is a sequence of BEERNAME elements:
  <BEERNAME>Bud</BEERNAME>
  <BEERNAME>Miller</BEERNAME> . . .

# Use of Braces

- When a variable name like $x, or an expression, could be text, we need to surround it by braces to avoid having it interpreted literally.
  - Example: <A>$x</A> is an A-element with value ”$x”, just like <A>foo</A> is an A-element with ”foo” as value.

# Use of Braces --- (2)

- But `return $x` is unambiguous.
- You cannot return an untagged string without quoting it, as `return "$x"`.

# LET Clauses

let <variable> := <expression>, . . .

- Value of the variable becomes the *sequence* of items defined by the expression.
- Note let does not cause iteration; for does.

# Example: LET

```
let $d := document(”bars.xml”)
let $beers := $d/BARS/BEER/@name
return
   <BEERNAMES> {$beers} </BEERNAMES>
```

- Returns one element with all the names of the beers, like:

```
<BEERNAMES>Bud Miller …</BEERNAMES>
```

# Order-By Clauses

- FLWR is really FLWOR: an order-by clause can precede the return.
- Form: order by <expression>
  - With optional ascending or descending.
- The expression is evaluated for each assignment to variables.
- Determines placement in output sequence.

# Example: Order-By

- List all prices for Bud, lowest first.

```
let $d := document("bars.xml")
for $p in
    $d/BARS/BAR/PRICE[@theBeer="Bud"]
order by $p
return $p
```

Generates bindings for $p to PRICE elements.

Order those bindings by the values inside the elements (automatic coersion).

Each binding is evaluated for the output. The result is a sequence of PRICE elements.

# Aside: SQL ORDER BY

- SQL works the same way; it's the result of the FROM and WHERE that get ordered, not the output.
- Example: Using R(a,b),

```
SELECT b FROM R
WHERE b > 10
ORDER BY a;
```

Then, the b-values are extracted from these tuples and printed in the same order.

R tuples with b>10 are ordered by their a-values.

# Predicates

- Normally, conditions imply existential quantification.
- Example: /BARS/BAR[@name] means “all the bars that have a name.”
- Example: /BARS/BEER[@soldAt = ”JoesBar”] gives the set of beers that are sold at Joe’s Bar.

# Example: Comparisons

- Let us produce the PRICE elements (from all bars) for all the beers that are sold by Joe's Bar.
- The output will be BBP elements with the names of the bar and beer as attributes and the price element as a subelement.

# Strategy

1. Create a triple for-loop, with variables ranging over all BEER elements, all BAR elements, and all PRICE elements within those BAR elements.
2. Check that the beer is sold at Joe's Bar and that the name of the beer and theBeer in the PRICE element match.
3. Construct the output element.

# The Query

```
let $bars = doc("bars.xml")/BARS
for $beer in $bars/BEER
for $bar in $bars/BAR
for $price in $bar/PRICE
where $beer/@soldAt = "JoesBar" and
  $price/@theBeer = $beer/@name
return <BBP bar = {$bar/@name} beer
  = {$beer/@name}>{$price}</BBP>
```

True if "JoesBar" appears anywhere in the sequence

# Strict Comparisons

- To require that the things being compared are sequences of only one element, use the Fortran comparison operators:
  - eq, ne, lt, le, gt, ge.
- Example: $beer/@soldAt eq ”JoesBar” is true only if Joe’s is the only bar selling the beer.

# Comparison of Elements and Values

- When an element is compared to a primitive value, the element is treated as its value, if that value is atomic.
- Example:

```
/BARS/BAR[@name="JoesBar"]/
PRICE[@theBeer="Bud"] eq "2.50"
```

is true if Joe charges $2.50 for Bud.

# Comparison of Two Elements

- It is insufficient that two elements look alike.
- Example:

```
/BARS/BAR[@name="JoesBar"]/
PRICE[@theBeer="Bud"] eq
/BARS/BAR[@name="SuesBar"]/
PRICE[@theBeer="Bud"]
```

is false, even if Joe and Sue charge the same for Bud.

# Comparison of Elements – (2)

- For elements to be equal, they must be the same, physically, in the implied document.
- Subtlety: elements are really pointers to sections of particular documents, not the text strings appearing in the section.

# Getting Data From Elements

- Suppose we want to compare the values of elements, rather than their location in documents.
- To extract just the value (e.g., the price itself) from an element *E*, use data(*E*).

# Example: data()

- Suppose we want to modify the return for "find the prices of beers at bars that sell a beer Joe sells" to produce an empty BBP element with price as one of its attributes.

```
return <BBP bar = {$bar/@name}
beer = {$beer/@name} price =
{data($price)} />
```

# Eliminating Duplicates

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

# Example: All the Distinct Prices

```
return distinct-values(
    let $bars = doc("bars.xml")
    return $bars/BARS/BAR/PRICE
)
```

Remember: XQuery is an expression language. A query can appear any place a value can.

# Effective Boolean Values

- The *effective boolean value* (EBV) of an expression is:
  1. The actual value if the expression is of type boolean.
  2. FALSE if the expression evaluates to 0, "" [the empty string], or () [the empty sequence].
  3. TRUE otherwise.

# EBV Examples

1. @name=”JoesBar” has EBV TRUE or FALSE, depending on whether the name attribute is ”JoesBar”.
2. /BARS/BAR[@name=”GoldenRail”] has EBV TRUE if some bar is named the Golden Rail, and FALSE if there is no such bar.

# Boolean Operators

- $E_1$ and $E_2$, $E_1$ or $E_2$, not($E$), apply to any expressions.
- Take EBV's of the expressions first.
- Example: not(3 eq 5 or 0) has value TRUE.
- Also: true() and false() are functions that return values TRUE and FALSE.

# Branching Expressions

- if ($E_1$) then $E_2$ else $E_3$ is evaluated by:
  - Compute the EBV of $E_1$.
  - If true, the result is $E_2$; else the result is $E_3$.
- Example: the PRICE subelements of $bar, provided that bar is Joe's.

```
if($bar/@name eq "JoesBar")
then $bar/PRICE else ()
```

Empty sequence. Note there is no if-then expression.

# Quantifier Expressions

some $x in $E_1$ satisfies $E_2$

1. Evaluate the sequence $E_1$.
2. Let $x (any variable) be each item in the sequence, and evaluate $E_2$.
3. Return TRUE if $E_2$ has EBV TRUE for at least one $x.

- Analogously:

every $x in $E_1$ satisfies $E_2$

# Example: Some

- The bars that sell at least one beer for less than $2.

```
for $bar in
    doc("bars.xml")/BARS/BAR
where some $p in $bar/PRICE
  satisfies $p < 2.00
return $bar/@name
```

Notice: where $bar/PRICE < 2.00 would work as well.

# Example: Every

- The bars that sell no beer for more than $5.

```
for $bar in
    doc("bars.xml")/BARS/BAR
where every $p in $bar/PRICE
  satisfies $p <= 5.00
return $bar/@name
```

# Document Order

- Comparison by document order: << and >>.
- Example: $d/BARS/BEER[@name="Bud"] << $d/BARS/BEER[@name="Miller"] is true iff the Bud element appears before the Miller element in the document $d.

# Set Operators

- union, intersect, except operate on sequences of nodes.
  - Meanings analogous to SQL.
  - Result eliminates duplicates.
  - Result appears in document order.

# XSLT

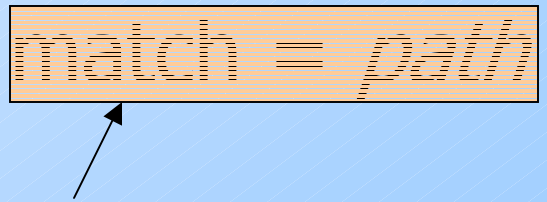
- XSLT (*extensible stylesheet language – transforms* ) is another language to process XML documents.
- Originally intended as a presentation language: transform XML into an HTML page that could be displayed.
- It can also transform XML -> XML, thus serving as a query language.

# XSLT Programs

- Like XML Schema, an XSLT program is itself an XML document.
- XSLT has a special namespace of tags, usually indicated by xsl:.

# Templates

- The xsl:template element describes a set of elements (of the document being processed) and what should be done with them.
- The form: <xsl:template match = *path* > … </xsl:template>



Attribute match gives an XPath expression describing how to find the nodes to which the template applies.

# Example: BARS Document -> Table

- In a running example, we'll convert the bars.xml document into an HTML document that looks like the Sells(bar, beer, price) relation.
- The first template will match the root of the document and produce the table without any rows.

# The Template for the Root

```
<xsl:template match = "/">
  <TABLE><TR>
    <TH>bar</th><TH>beer</th>
    <TH>price</th></tr>
</table>
</xsl:template>
```

Template matches only the root.

Output of the template is a table with the attributes in the header row, no other rows.

Needs to be fixed. As is, there is no way to insert rows.

# Outline of Strategy

1. Inside the HTML for the table is xsl:apply-templates to extract data from the document.
2. From each BAR, use an xsl:variable *b* to remember the bar name.
3. xsl:for-each PRICE subelement, generate a row, using *b*, and xsl:value-of to extract the beer name and price.

# Recursive Use of Templates

- An XSLT document usually contains many templates.
- Start by finding the first one that applies to the root.
- Any template can have within it <xsl:apply-templates/>, which causes the template-matching to apply recursively from the current node.

# Apply-Templates

- Attribute select gives an XPath expression describing the subelements to which we apply templates.
- Example: <xsl:apply-templates select = ”BARS/BAR” /> says to follow all paths tagged BARS, BAR from the current node and apply all templates there.

# Example: Apply-Templates

```
<xsl:template match = "/">
  <TABLE><TR>
     <TH>bar</th><TH>beer</th>
     <TH>price</th></tr>
  <xsl:apply-templates select =
     "BARS" />
  </table>
</xsl:template>
```

# Extracting Values

- <xsl:value-of select = XPath expression /> produces a value to be placed in the output.
- Example: suppose we are applying a template at a BAR element and want to put the bar name into a table.

```
<xsl:value-of select = "@name" />
```

# Variables

- We can declare x to be a variable with <xsl:variable name = ”x” />.
- Example:

```
<xsl:variable name = ”bar”>
  <xsl:value-of select = ”@name” />
</xsl:variable>
```

within a template that applies to BAR elements will set variable bar to the name of that bar.

# Using Variables

- Put a $ in front of the variable name.
- Example: <TD>$bar</td>

# Completing the Table

1. We'll apply a template at each BAR element.
2. This template will assign a variable b the value of the bar, and iterate over each PRICE child.
3. For each PRICE child, we print a row, using b, the theBeer attribute, and the PRICE itself.

# Iteration

- <xsl:for-each select = Xpath expression>

  ...

  </xsl:for-each>

  executes the body of the for-each at each child of the current node that is reached by the path.

# The Template for BARS