XML

csc343, winter 2011

Diane Horton
University of Toronto

Based on slides by Jeff Ullman

Two kinds of XML document

- *Well-Formed XML*
  - Just need to use proper nesting.
  - Can invent your own tags.
  - Any tag can go anywhere.
- *Valid XML*
  - Can invent tags, but you define what they are and where they can go.
  - A "DTD" (document type definition) specifies these rules.

Well-Formed XML

- Start the document with a *declaration*, surrounded by <?xml … ?> .
- Normal declaration is:

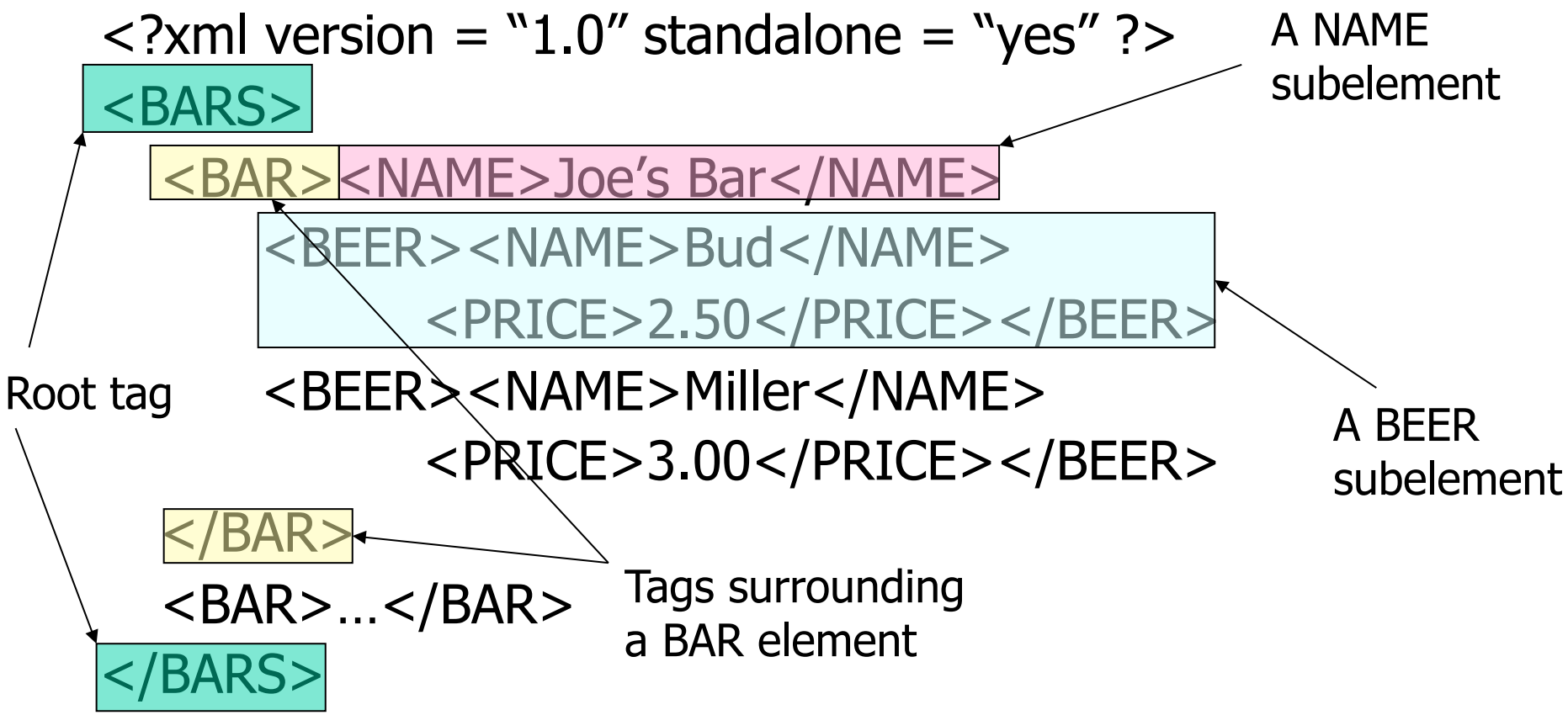
```
<?xml version = ”1.0” standalone = ”yes” ?>
```

  - “standalone” means no DTD is provided.
- Balance of document is a *root tag* surrounding nested tags.

Tags

- Tags are normally matched pairs, e.g., <FOO> … </FOO>.
- Unmatched tags are also allowed, e.g., <FOO/>
- Tags may be nested arbitrarily.
- XML tags are case-sensitive.

Example: Well-Formed XML



Nesting rule for tags must be obeyed

Attributes

- An opening tag can have attribute name-value pairs within it. Example: <BEER name="Bud" price=2.50/>
- The pairs are separated by blanks.
- Here, we put all the info in the attributes, so the tag became empty.
- Could have some info in attribute values and some in the tag body.
- It's a design decision.

Checking your XML

- http://validator.w3.org
- xmllint command on cdf.
  By default, checks if well-formed.
- --debug
  Outputs an annotated tree of the parsed document.

Problems with merely well-formed XML

- If a program will process XML, good to know things like:
  - What tags are allowed
  - What order, nesting
  - What attributes for each tag
  - What's mandatory or optional
- A DTD specifies exactly this.

Valid XML with DTDs

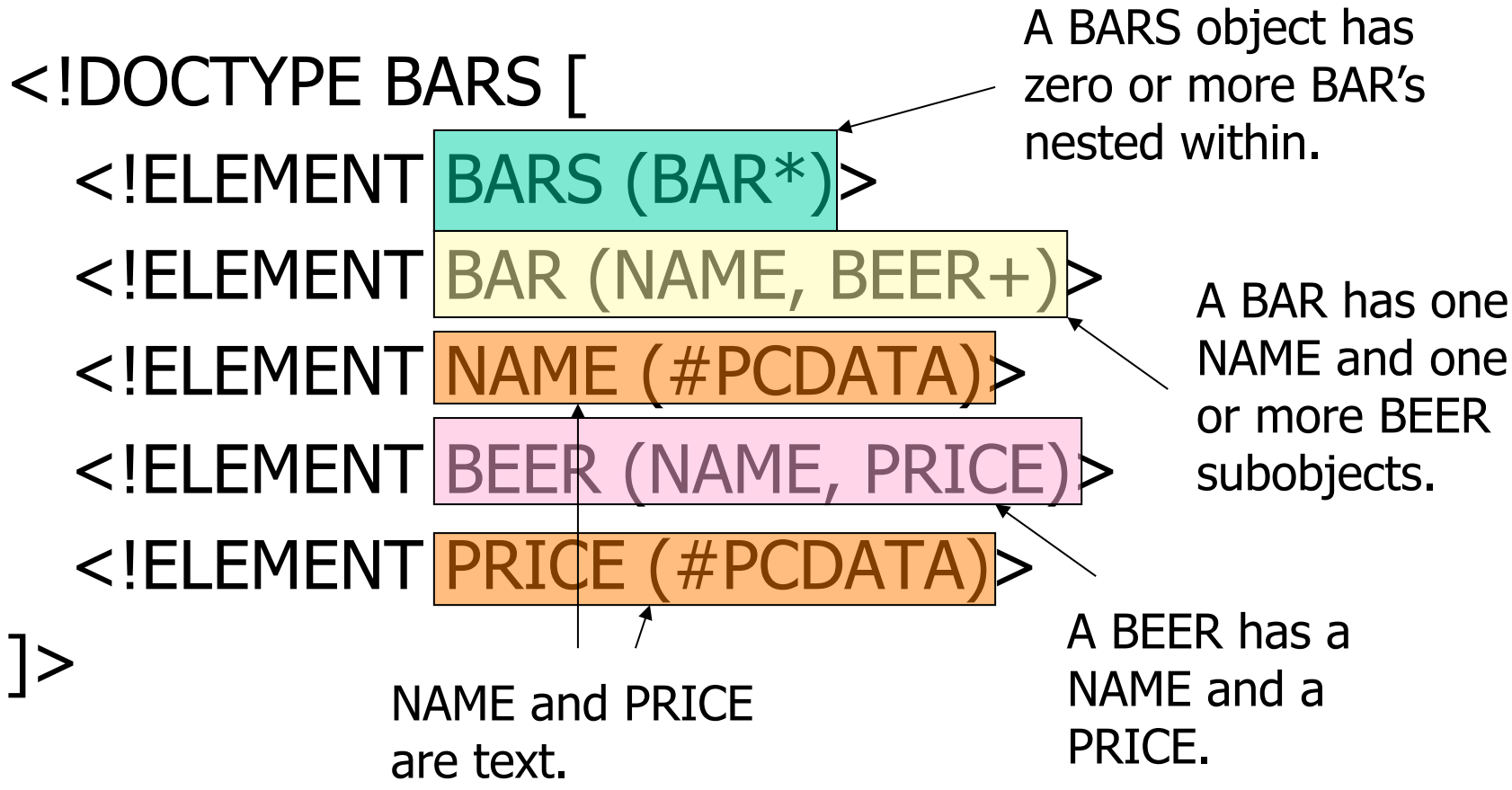
DTD Structure

```
<?xml version = ”1.0” standalone = ”yes” ?>
<!DOCTYPE <root tag> [
  <!ELEMENT <name>(<components>)>
  ... more elements ...
]>
<actual xml content goes here>
```

DTD Elements

- The description of an element consists of its name (tag), and a parenthesized description of any nested tags.
  - Includes order of subtags and their multiplicity.
- Leaves (text elements) have `#PCDATA` ("Parsed Character DATA") instead of nested tags.

Example: DTD


<!DOCTYPE BARS [
<!ELEMENT BARS (BAR*)>
<!ELEMENT BAR (NAME, BEER+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT BEER (NAME, PRICE)>
<!ELEMENT PRICE (#PCDATA)>
]>
A BARS object has zero or more BAR's nested within.
A BAR has one NAME and one or more BEER subobjects.
A BEER has a NAME and a PRICE.
NAME and PRICE are text.

Element Descriptions

- They symbol “,” indicates that subtags must appear in the order shown.
- The symbol | indicates alternatives.
- A subtag may be followed by a symbol to indicate its multiplicity.
  - * = zero or more.
  - + = one or more.
  - ? = zero or one.
- Brackets can be used.

Example: Element Description

- A name is: an optional title (e.g., "Prof."), a first name, and a last name, in that order, or it is an IP address:

```
<!ELEMENT NAME (
  (TITLE?, FIRST, LAST) | IPADDR
)>
```

Use of DTD's

1. Set standalone = "no".
2. Then either:
   a) Include the DTD as a preamble of the XML document, or
   b) Follow DOCTYPE and the <root tag> by SYSTEM and a path to the file where the DTD can be found.

Example: (a)

```
<?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> …
</BARS>
```

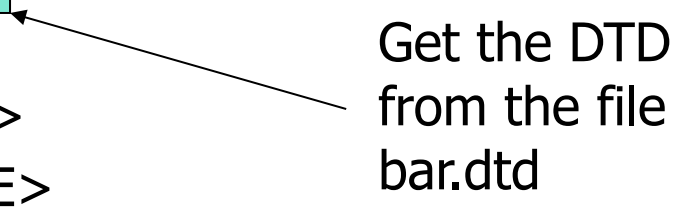
The DTD

The document

Example: (b)

- Assume the BARS DTD is in file bar.dtd.

```
<?xml version = "1.0" standalone = "no" ?>
<!DOCTYPE BARS SYSTEM "bar.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> …
</BARS>
```



Attributes

- In a DTD, we specify each attribute of an element like this:

<!ATTLIST *E aName aType aOptionality*>

Example: Attribute definition

- Bars can have an attribute `kind`, a character string describing the bar.

```
<!ELEMENT BAR (NAME BEER*)>
<!ATTLIST BAR kind CDATA #IMPLIED>
```

Character string type; no tags

Attribute is optional. Opposite: #REQUIRED

Example: Attribute Use

- In a document that allows BAR tags, we might see:

```
<BAR kind = ”sushi”>
  <NAME>Homma’s</NAME>
  <BEER><NAME>Sapporo</NAME>
        <PRICE>5.00</PRICE></BEER>
  ...
</BAR>
```

“Keys” and “foreign keys”

The closest thing to a key

- To create unique identifiers:
  - Give an element *E* an attribute *A* of type `ID`.
  - When using tag *<E>*, give its attribute *A* a unique value.
- Example:

```
<E A="xyz">
```

We need foreign keys too

- In HTML, we make pointers using NAME = ”foo” and HREF = ”#foo”.
- XML attributes also can be pointers from one object to another.
- Allows the structure of an XML document to be a general graph, rather than just a tree.

The closest thing to a foreign key

- To allow elements of type *F* to refer to another element with an `ID` attribute, give *F* an attribute of type `IDREF`.
- Or, let the attribute have type `IDREFS`, so the *F*-element can refer to any number of other elements.

Example: ID's and IDREF's

- A new BARS DTD includes both BAR and BEER subelements.
- BARS and BEERS have ID attributes `name`.
- BARS have SELLS subelements, consisting of a number (the price of one beer) and an IDREF `theBeer` leading to that beer.
- BEERS have attribute `soldBy`, which is an IDREFS leading to all the bars that sell it.

The 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>
]>
```

Example: A Document

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

Empty Elements

- We can do all the work of an element in its attributes.
  - Like BEER in previous example.
- Another example: SELLS elements could have attribute `price` rather than a value that is a price.

Example: Empty Element

- In the DTD, declare:

```
<!ELEMENT SELLS EMPTY>
   <!ATTLIST SELLS theBeer IDREF #REQUIRED>
   <!ATTLIST SELLS price CDATA #REQUIRED>
```

- Example use:

```
<SELLS theBeer = ”Bud” price = ”2.50” />
```

Note exception to
“matching tags” rule

Checking validity of your XML

- We saw that xmllint can determine if your xml is well-formed.
- It can also check validity.
- --valid
  Determine if the document is a valid with respect to its DTD.
- --schema SCHEMA
  Determine if the document is valid with respect to its XML Schema . . .

XML Schema

Introduction

- XML Schema is a more powerful way to describe the structure of XML documents.
- XML Schema Definitions (XSD) are themselves XML documents.
  - They describe “elements” and
  - the things doing the describing are themselves “elements.”

Structure of an XML-Schema Document

```
<? xml version = … ?>
<xs:schema xmlns:xs =
  ”http://www.w3.org/2001/XMLschema”>
                . . .
</xs:schema>
```

So uses of ”xs” within the schema element refer to tags from this namespace.

Defines ”xs” to be the *namespace* described in the URL shown. Any string in place of ”xs” is OK.

- There is much more to XML Schema than we will cover.
- As a start, you can treat the previous slide as a template for all XSD files.
- Now let's look at what goes inside the xs:schema tag:
  - Elements (because this is an XML file!)
  - They define the elements, attributes etc. that are allowed in xml docs that are instances of the schema.

xs:element

- Defines what elements are allowed in xml docs that are instances of this schema.
- Has attributes:
  1. name = the tag-name of the element being defined.
  2. type = the type of the element.
     - Could be an XML-Schema type, e.g., xs:string.
     - Or the name of a type defined in the document itself.

Example: xs:element

```
<xs:element
      name = ”ADDRESS”
      type = ”xs:string” />
```

- Describes elements such as

  <ADDRESS>98 Pine St.</ADDRESS>

Built-in and Complex Types

- Built-in types:
  - We've seen `xs:string`
  - There is also `xs:float`, `xs:boolean`, `xs:dateTime`, etc.
- We can also define "complex" types, using `xs:complexType`
  - For defining elements that contain elements.
  - Perhaps better called "compound" types.

xs:complexType

- Used to define a new type for elements that contain subelements.
  - Attribute name gives a name to the type.
- To make it contain, e.g., an ordered sequence of 3 elements
  - Give it an xs:sequence subelement.
  - Give that 3 xs:element subelements.
  - Can use minOccurs and maxOccurs attributes to control the number of occurrences of each xs:element.

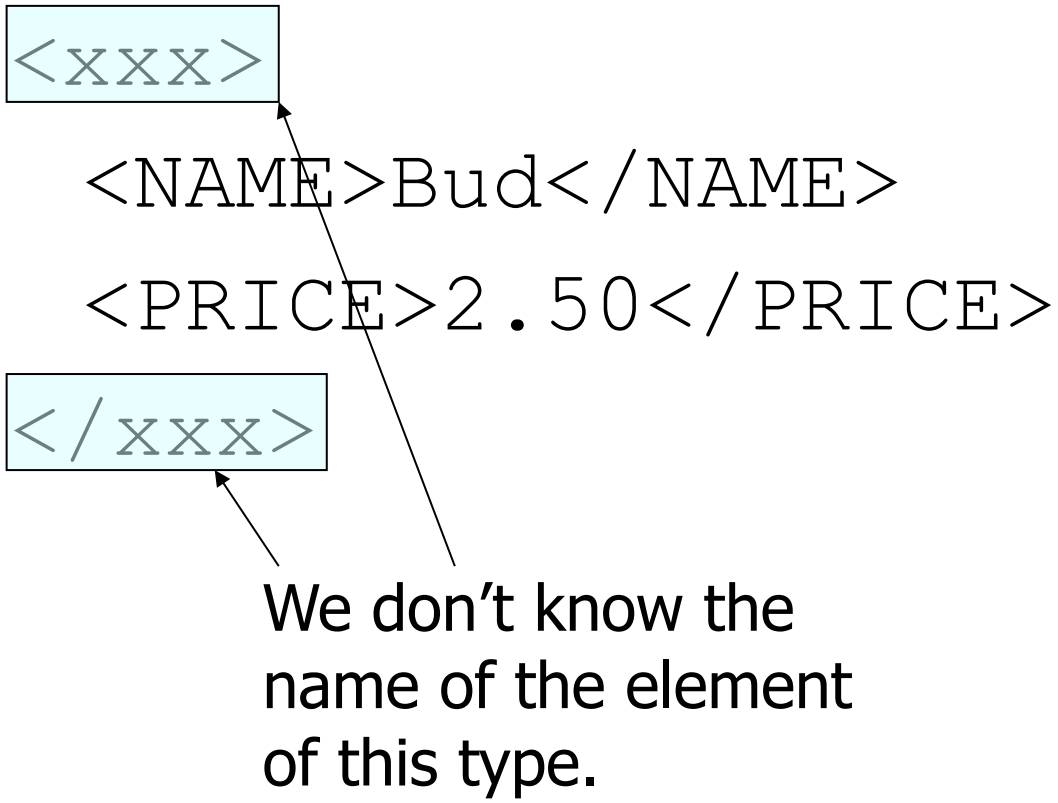
Example: a Type for Beers

```
<xs:complexType name = "beerType">
  <xs:sequence>
    <xs:element name = "NAME"
      type = "xs:string"
      minOccurs = "1" maxOccurs = "1" />
    <xs:element name = "PRICE"
      type = "xs:float"
      minOccurs = "0" maxOccurs = "1" />
  </xs:sequence>
</xs:complexType>
```

Exactly one occurrence

Like ? in a DTD

An instance of type beerType



Example: a Type for Bars

```
<xs:complexType name = "barType">
  <xs:sequence>
    <xs:element name = "NAME"
      type = "xs:string"
      minOccurs = "1" maxOccurs = "1" />
    <xs:element name = "BEER"
      type = "beerType"
      minOccurs = "0" maxOccurs =
                  "unbounded" />
  </xs:sequence>
</xs:complexType>
```

Like * in a DTD

All the options for complex types

- xs:sequence
  - The elements must occur *in order.*
  - Control number of occurrences with minOccurs (default 1) and maxOccurs attributes (default 1; can be "unbounded").
- xs:all
  - The elements can occur in *any order*.
  - Each can occur 0 or 1 times.
- xs:choice
  - Exactly one of the elements must occur.
  - Can use maxOccurs to allow it to repeat.

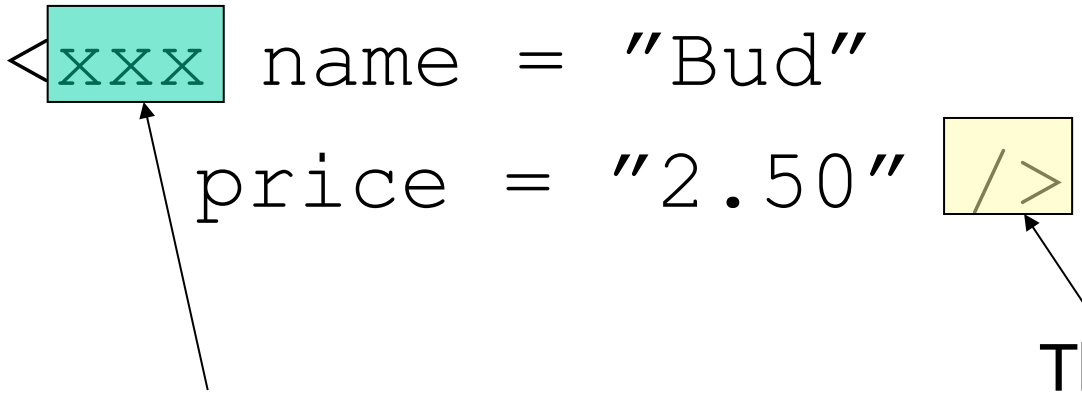
xs:attribute

- Can be used within a complex type to indicate attributes of elements of that type.
- attributes of xs:attribute:
  - name and type as for xs:element.
  - use = ”required” or ”optional”.
  - default = *<some default value>*

Example: xs:attribute

```
<xs:complexType name = ”beerType”>
  <xs:attribute name = ”name”
     type = ”xs:string”
     use = ”required” />
  <xs:attribute name = ”price”
     type = ”xs:float”
     use = ”optional” />
</xs:complexType>
```

An instance of beerType



Restrictions on simple types

- You can define new types that are restrictions on xs:string, xs:float, etc.
- Can restrict numeric values to a range.
- Can restrict any kind of values to a given list of options (an “enumeration”).
- Use xs:simpleType

Example: Prices in Range [1,5)

```
<xs:simpleType name = ”price”>
  <xs:restriction
     base = ”xs:float”
     minInclusive = ”1.00”
     maxExclusive = ”5.00” />
</xs:simpleType>
```

Example: license Attribute for BAR

```
<xs:simpleType name = ”license”>
  <xs:restriction base = ”xs:string”>
    <xs:enumeration value = ”Full” />
    <xs:enumeration value = ”Beer only” />
    <xs:enumeration value = ”Sushi” />
  </xs:restriction>
</xs:simpleType>
```

Restrictions

- Attribute base gives the simple type to be restricted, e.g., xs:integer.
- {min, max}{Inclusive, Exclusive} are four attributes that can give a lower or upper bound on a numerical range.
- xs:enumeration is a subelement with attribute value that defines the options for an enumerated type.

Keys and foreign keys

Better than in DTDs

- You can require uniqueness within restricted related spots in the xml document.
  - DTDs: uniqueness is across the entire document!
- You can specify that a "keyref" refers to a particular type of element.
  - DTDs: good enough if refers to *any* element!

Keys in XML Schema

- An xs:element can have an xs:key subelement.
- Meaning: within this element, all subelements reached by a certain *selector* path will have unique values for a certain combination of *fields*.
- Example: within one BAR element, the name attribute of a BEER element is unique.

Example: Key

```
<xs:element name = "BAR" … >
    . . .
  <xs:key name = "barKey">
     <xs:selector xpath = "BEER" />
     <xs:field xpath = "@name" />
  </xs:key>
    . . .
</xs:element>
```

An @ indicates an attribute rather than a tag.

XPath is a query language for XML. All we need to know here is that a path is a sequence of tags separated by /.

Foreign Keys

- An xs:keyref subelement within an xs:element says that within this element, certain values (defined by selector and field(s), as for keys) must appear as values of a certain key.

Example: Foreign Key

- Suppose that we have declared that subelement NAME of BAR is a key for BARS.
  - The name of the key is barKey.
- We wish to declare DRINKER elements that have FREQ subelements. An attribute bar of FREQ is a foreign key, referring to the NAME of a BAR.

Example: Foreign Key in XML Schema

```
<xs:element name = ”DRINKERS”
    . . .
  <xs:keyref name = ”barRef”
     refers = ”barKey”
     <xs:selector xpath =
          ”DRINKER/FREQ” />
     <xs:field xpath = ”@bar” />
  </xs:keyref>
</xs:element>
```

Query Languages for XML

csc343, winter 2011

Diane Horton

University of Toronto

Based on slides by Jeff Ullman; also some by Ramona Truta

Data model

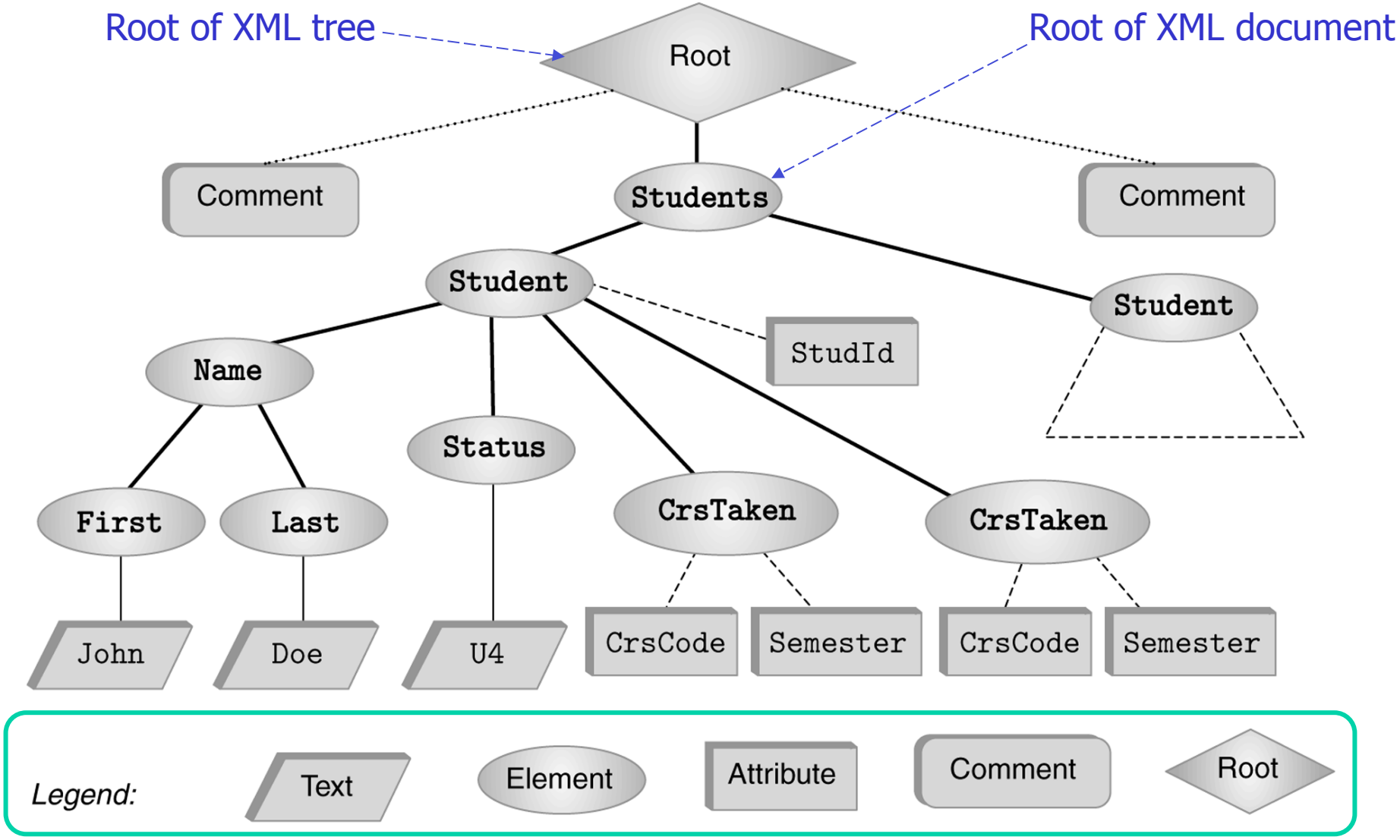
XML document as a tree

- Root is a node which doesn't correspond to anything in the document.
- Internal nodes are elements of the document.
- Leaves are:
  - attributes
  - text nodes (for the body of an element)
  - comments
  - or other things that we won't discuss (processing instructions, …)

Example document

```
<?xml  version="1.0" ?>
<!-- Some  comment -->
<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>
<!-- Some other comment -->
```

Document tree



XPath

Main idea

- Goal of a query is to find items you want from a document.
  Example: all course codes.
- Describe what you want by defining path(s) through the tree to get to it.
- Example:
  root →Student →CrsTaken→CrsCode attribute
- Use path expressions to express this.

Results

- Result of a path expression is a *sequence of items*.
- An *item* is either:
  - A primitive value, e.g., integer or string.
  - A node.

Homo or Hetero

- Depending on the query, the result could be heterogeneous. Eg:

```
13
<stylist> …. </stylist>
@age=26
```

- But we often write queries to give homogenous results.
  Eg: find the name of all movies.

```
<MOVIE name="The Town"/>
<MOVIE name="Hereafter"/>
```

Running XPath queries

- We'll use galax on cdf.
  firefox /usr/share/doc/galax-doc/manual/manual.html
- At the command line, type:

  `galax-run` *`query.xq`*
- File *`query.xq`* must have the form:

  `fn:doc("`*`doc.xml`*`")`*`path-expn`*

  where
  - *`doc.xml`* is an xml file, and
  - *`path-expn`* is an Xpath path expression

Principal Kinds of Nodes

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

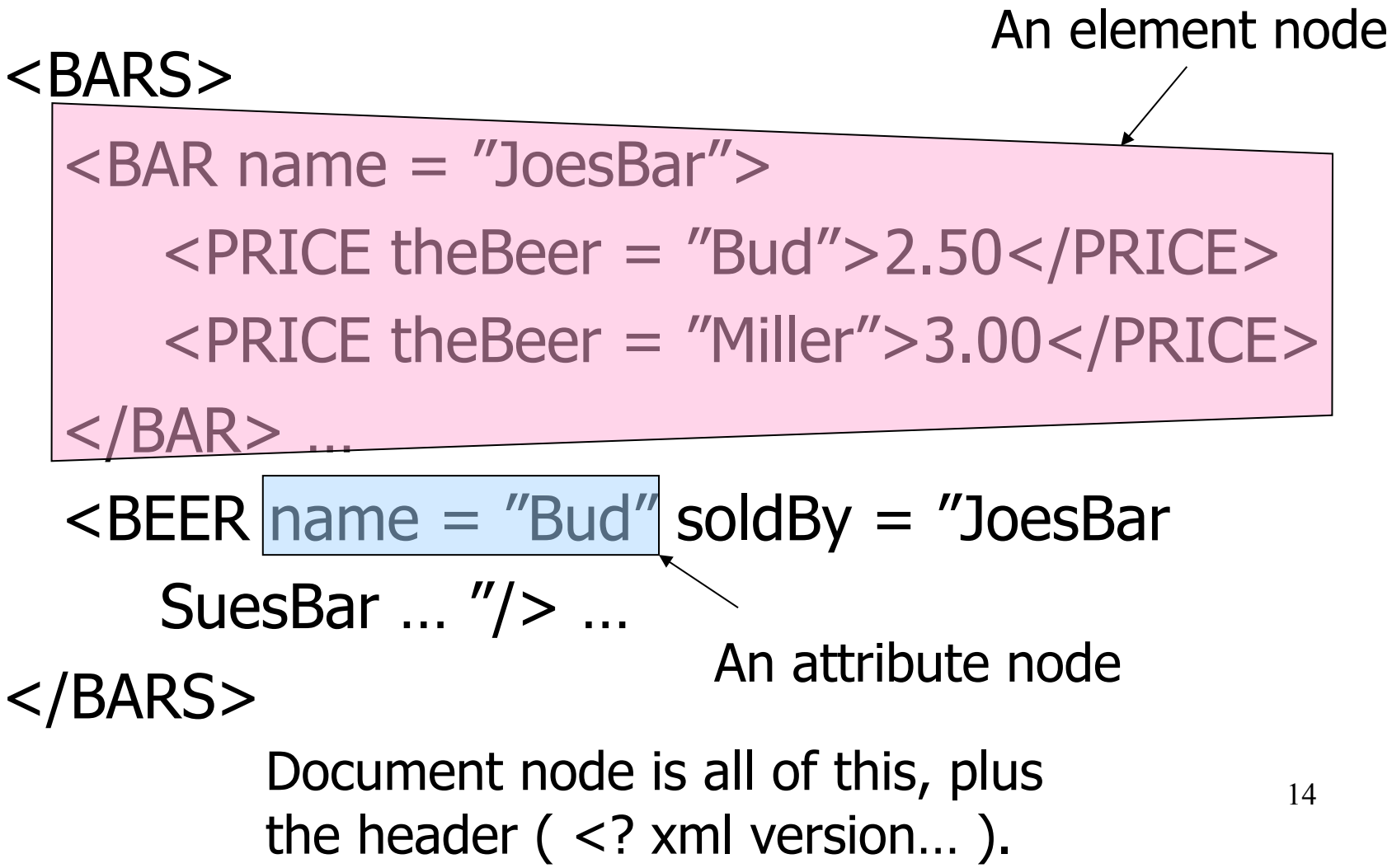
Document Nodes

- `fn:doc("doc.xml")` is a function call.
- (The full name of this function is `document`, but we can use `doc` for short.)
- It returns a document node that is the root of the whole tree.
- The XPath expression is evaluated with respect to that document node.

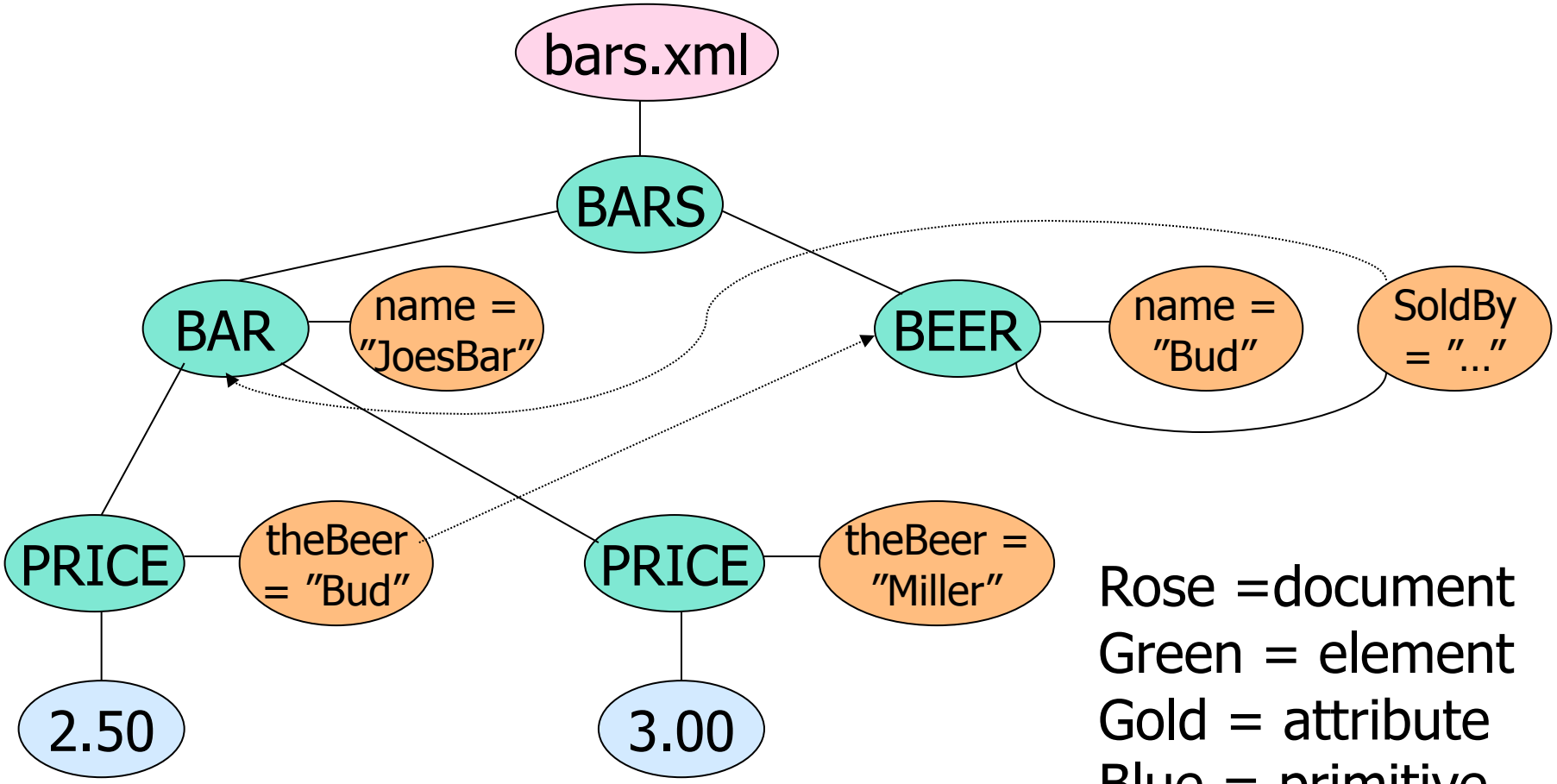
DTD for Running Example

```
<!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



Tree for this document



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

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

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 in order.

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

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

One item, the BARS element

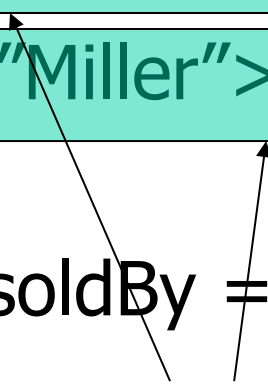
Example: /BARS/BAR

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

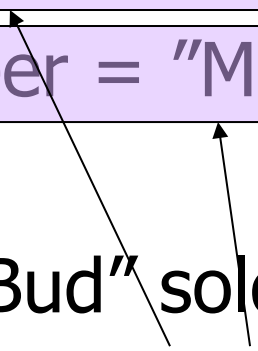
Using attributes

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

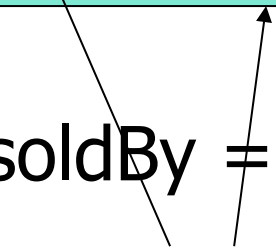
- Until now, all item sequences have been sequences of elements.
- When a path expression ends in an attribute, the result is typically a sequence of values of primitive type, such as strings in the previous example.

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>
```

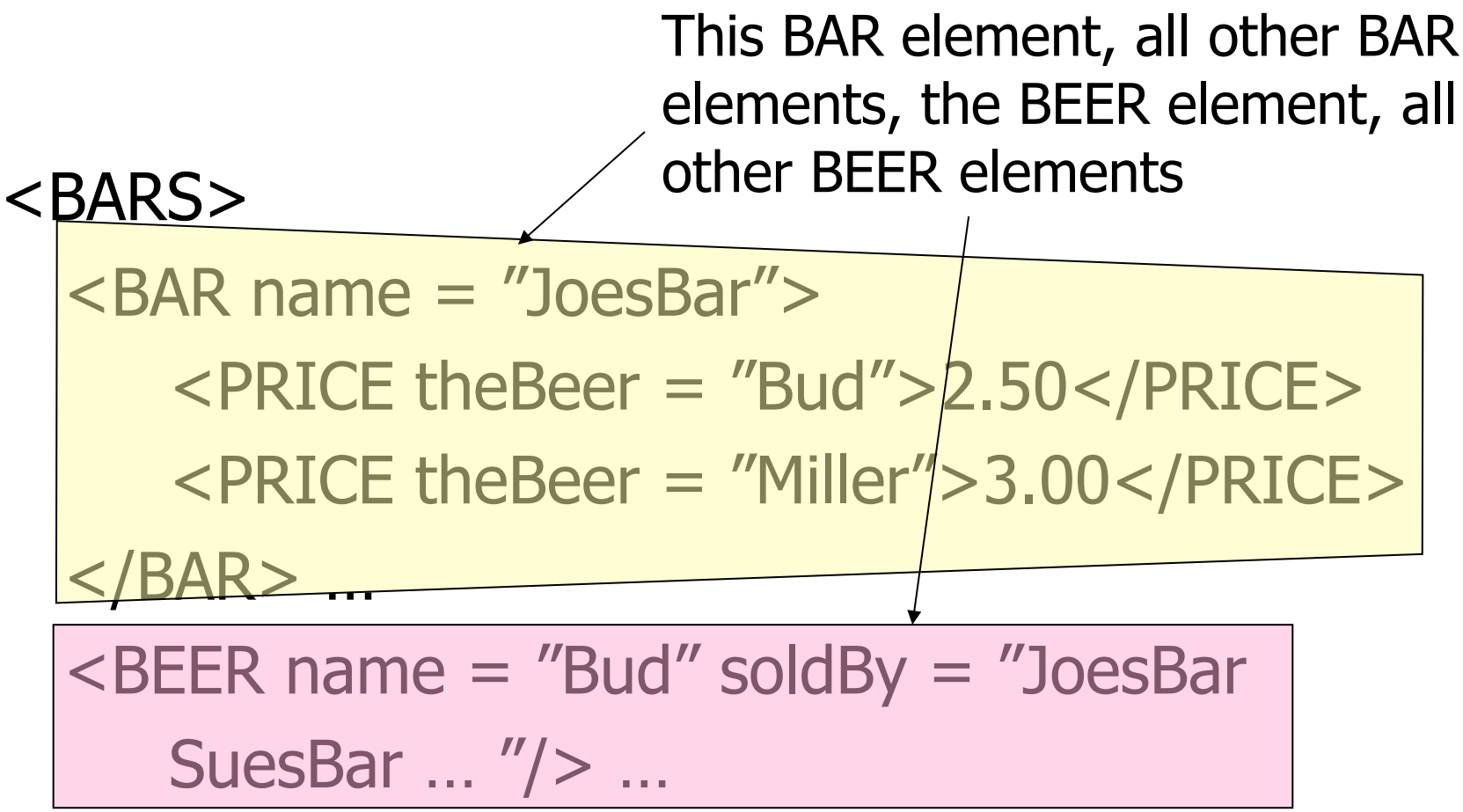


These PRICE elements and any other PRICE elements in the entire document

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/*



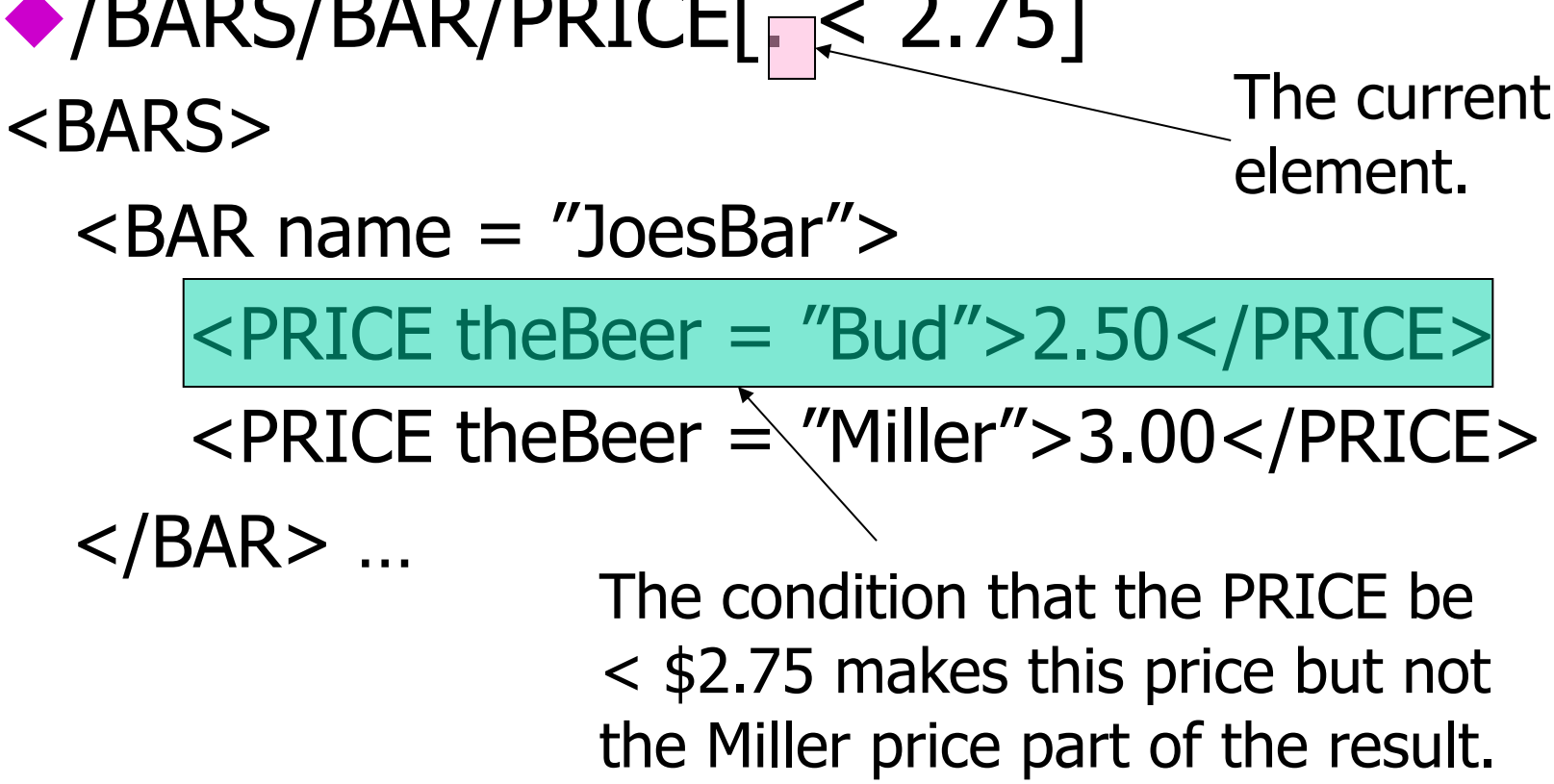
Selection conditions

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.

More general “axes”

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:
  - parent:: = parent(s) of the current node (s).
  - descendant-or-self:: = the current node (s) and all descendants.
    - Note: // is really shorthand for this axis.
  - ancestor::, ancestor-or-self, etc.
  - self (the dot).

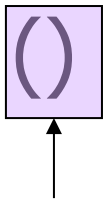
Xquery query language

XQuery

- Extends Xpath; has power similar to SQL.
- Uses the same data model
  - Document is a tree
  - 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.
  - Like RA; unlike SQL

Item sequences are flattened

- XQuery will sometimes form nested sequences.
- These are always flattened.
- Example: (1 2 () (3 4)) = (1 2 3 4).



Empty sequence

Syntax: FLWR expressions

Simplified syntax:

for <var> in <expn>
let <var> := <expn>

Any number of these, in any order

where <expn>

optional

return <expn>

Notes:

- Variables begin with $
- Keywords are case sensitive

Semantics of FLWR Expressions

- Each for creates a loop.
- let defines a variable.
- At each iteration of the nested loops, if any, evaluate the where clause.
- If the where clause returns TRUE, append the value of the return clause to the output and continue.
  - return does not end evaluation of the FLWR expression.

FOR Clauses

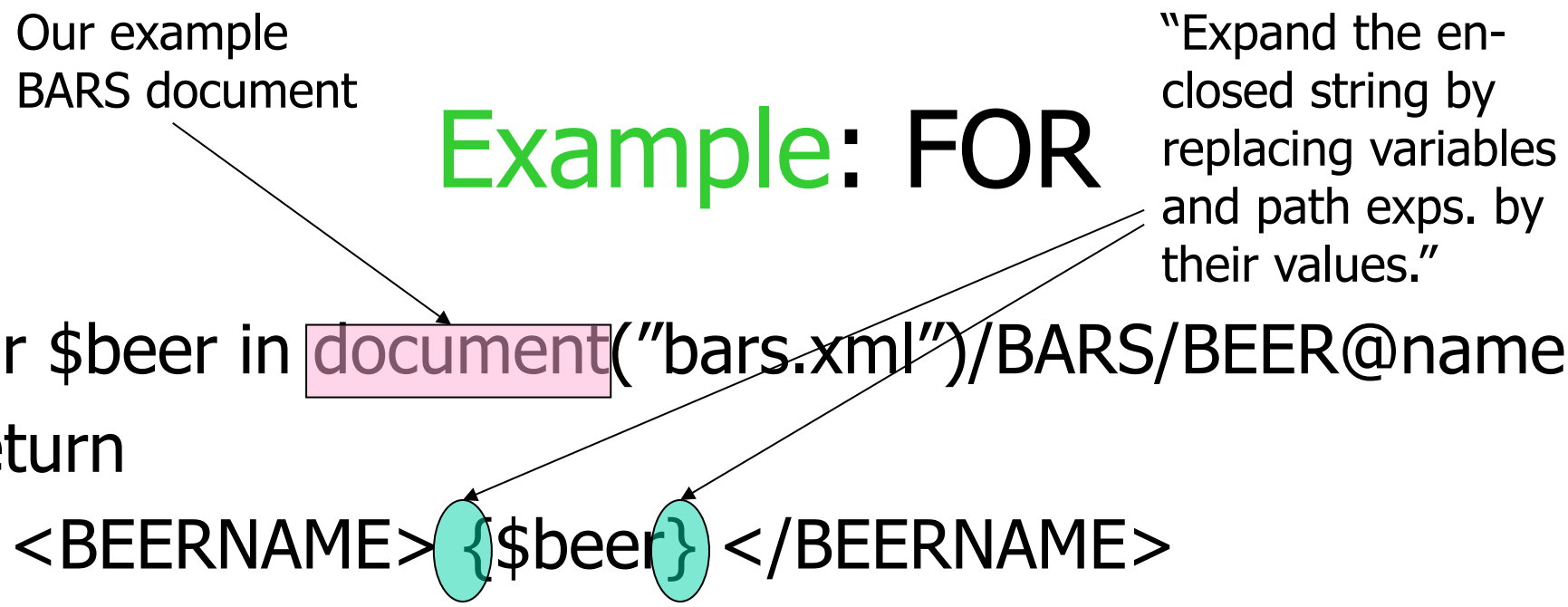
for <variable> in <expression>, . . .

- The expression is evaluated.
- The for-variable takes on each item in the resulting sequence, in turn.
- Whatever follows the 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 or an expression could be text, we need to surround it by braces to avoid having it interpreted literally.
- Example: <A>$x</A> is considered to be an A-element with value ”$x”, just like <A>foo</A> is an A-element with value ”foo”.

Use of Braces --- (2)

- But with `return $x` the `$x` is evaluated.
- To prevent evaluation, quote it: `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.

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

Keep in Mind: 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.

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

- Create a triple for-loop, with variables ranging over all BEER elements, all BAR elements, and all PRICE elements within those BAR elements.
- Check that the beer is sold at Joe's Bar and that the name of the beer and theBeer in the PRICE element match.
- 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:
  - The actual value if the expression is of type boolean.
  - FALSE if the expression evaluates to 0, "" [the empty string], or () [the empty sequence].
  - TRUE otherwise.

EBV Examples

- @name=”JoesBar” has EBV TRUE or FALSE, depending on whether the name attribute is ”JoesBar”.
- /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

- Evaluate the sequence E_1.
- Let $x (any variable) be each item in the sequence, and evaluate E_2.
- 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.