XML

- Semi-structured data
- XML
- XML type specification languages:
  - Document Type Definitions (DTD)
  - XML Schema
- Query languages:
  - XPath
  - XQuery

Semistructured Data

- Another data model, based on trees.
- Self-describing:
  - The data implicitly carries information about what its schema is.
  - May only carry the names of attributes (so possibly untyped), and has a lower degree of organization than the data in a relational database.
  - May have no associated schema (i.e. may be schema-less)
- Motivation:
  - flexible representation of data.
  - sharing of *documents* among systems and databases.

Structured vs. Semi-Structured Data

- Relational data is structured
  - Data follows a given schema
  - Data is stored in database tables
- Consider a schema for patients in a clinic:
  - Some fields are highly structured, e.g., name, address
  - Some fields are not structured, e.g., descriptions
  - Many fields are optional, e.g., X-ray

Applications of Semistructured Data

- Data exchange
  - E.g. two enterprises may want to exchange data (such as buyers and sellers)
- Information integration
  - E.g. want to “merge” or query two databases.

Graphs of Semistructured Data

- A database of semistructured data can be considered as a rooted tree (or graph) made up of nodes and arcs.
  - Tree gives a hierarchical structure to the data
- Nodes = objects
- Arcs give relations between nodes.
- Labels on arcs (like attribute names).
- Atomic values at leaf nodes.
- Flexibility: no restriction on
  - Labels out of a node.
  - Number of successors with a given label.

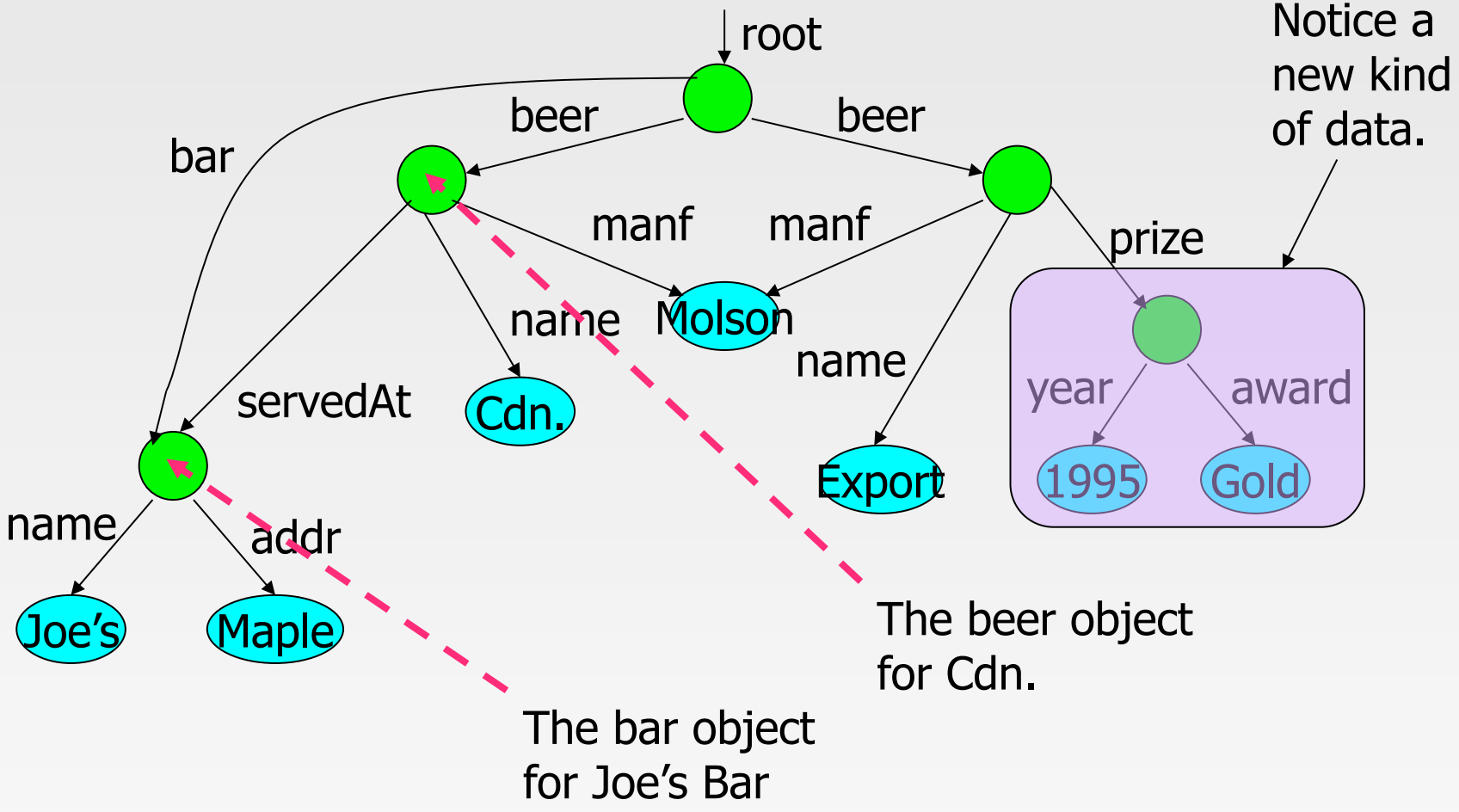
XML

- XML = *Extensible Markup Language*.
  - A standard adopted in 1998 by the W3C (World Wide Web Consortium)
- Looks like HTML code
- While HTML uses tags for formatting (e.g., “italic”), XML uses tags for semantics (e.g., indicating “this is an address” or “this is a title”).
- Key idea: create tag sets for a domain (e.g., genomics), and translate all data into properly tagged XML documents.

XML (continued)

- Optional mechanisms for specifying document structure (schema)
  - DTD: the Document Type Definition Language, part of the XML standard
  - XML Schema: a more recent specification built on top of XML
    - Discussed only briefly
- Query languages for XML
  - XPath: a lightweight, incomplete, language
  - XQuery: a full-blown language
  - XSLT: document transformation language
    - (not covered)

Example: Data Graph



XML: Motivation

- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML-based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Definition)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, these may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Example: Well-Formed XML

```
<?xml version = “1.0” standalone = “yes” ?>
<BARS>
    <BAR><NAME>Joe’s Bar</NAME>
        <BEER><NAME>G.I. Lager</NAME>
            <PRICE>2.50</PRICE></BEER>
        <BEER><NAME>Export</NAME>
            <PRICE>3.00</PRICE></BEER>
    </BAR>
    <BAR> …  </BAR>
      …
</BARS>
```

Comparison with Relational Data

- Disadvantages (compared to relational model):
  - Inefficient: tags, which in effect represent schema information, are repeated
  - Access: data is structured hierarchically.
- Advantages:
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Flexible, non-rigid format: tags can be added
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Well-Formed and Valid XML

There are two different modes of use of XML:

- *Well-Formed XML* allows you to invent your own tags.
  - No predefined schema
- *Valid XML* conforms to a certain DTD.
  - The DTD describes allowable tags and their nesting.
  - But still fairly flexible – e.g. may allow optional or missing fields

Well-Formed XML

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

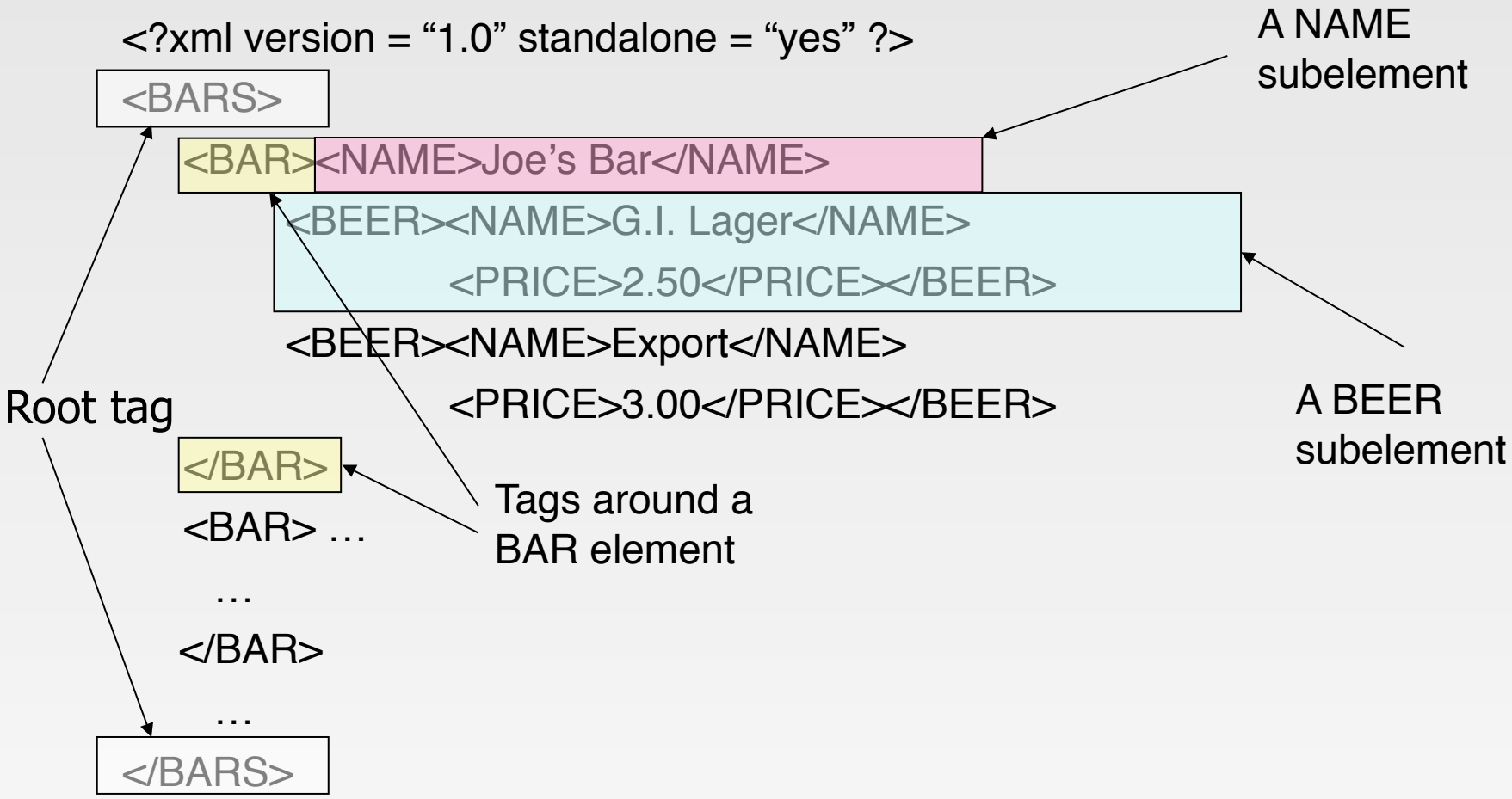
  <?xml version = ”1.0” standalone = ”yes” ?>

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

Tags

- Tags are normally matched pairs, as <FOO> … </FOO>.
- Unmatched tags also allowed, as <FOO/>
- Tags may be nested arbitrarily.
  - Think of record structures
- XML tags are case-sensitive.
- Tags may also have attributes (later).

Example: Well-Formed XML



Attributes

- Elements can have attributes

```
<BAR NAME = “Joes Bar”>
        <BEER><NAME>Export</NAME>
                <PRICE>2.50</PRICE></BEER>
        …
</BAR>
```

- Attributes are given by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can occur only once.

```
<account acct-type = “chequeing” monthly-fee = “5”>
```

Valid XML: DTD Structure

- DTD: "*Document Type Definition*"
- A DTD gives a specification of a *schema* for an XML document
- Describes:
  - kinds of elements and
  - how they can be nested

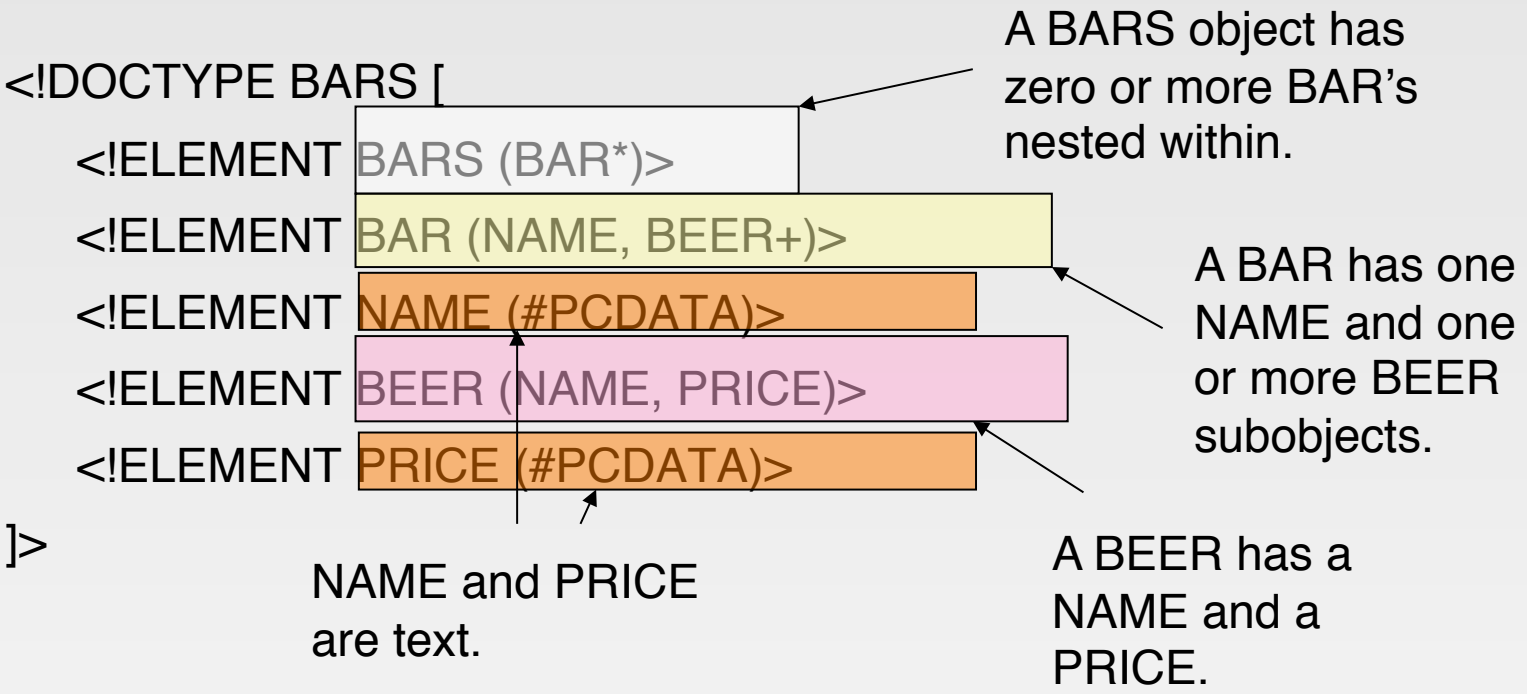
- Overall structure:

```
<!DOCTYPE <root tag> [
        <!ELEMENT <name>(<components>)>
        . . . more elements . . .
]>
```

DTD Elements

- The description of an element consists of its name (tag), and a parenthesized description of any nested tags.
  - Includes the order of subtags and their multiplicity.
- Leafs (text elements) have #PCDATA (*Parsed Character DATA*) in place 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

- Subtags must appear in order shown.
- A tag may be followed by a symbol to indicate its multiplicity.
  - * = zero or more.
  - + = one or more.
  - ? = zero or one.
- Symbol “|” can connect alternative sequences of tags.

Example: Element Description

- A person's name can be defined as an optional title (e.g., "Ms."), 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. 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)>
]>
```

The DTD

The document

```
<BARS>
    <BAR><NAME>Joe's Bar</NAME>
        <BEER><NAME>Export</NAME> <PRICE>2.50</PRICE></BEER>
        <BEER><NAME>Gr.Is.</NAME> <PRICE>3.00</PRICE></BEER>
    </BAR>
    <BAR> …
</BARS>
```

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>Export</NAME>
            <PRICE>2.50</PRICE></BEER>
        <BEER><NAME>Gr.Is.</NAME>
            <PRICE>3.00</PRICE></BEER>
    </BAR>
    <BAR> …
</BARS>
```

Get the DTD from the file bar.dtd

Attributes

- Opening tags in XML can have *attributes*.
- In a DTD, declare using

  <!ATTLIST *element-name attr-name type* … >

  declares attributes for element *element-name*, along with their datatypes.

Example: Attributes

- 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

Optional attribute opposite: #REQUIRED

Example: Attribute Use

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

```
<BAR kind = ”sushi”>
    <NAME>Shyun</NAME>
    <BEER><NAME>Sapporo</NAME>
        <PRICE>5.00</PRICE>
    </BEER>
    ...
</BAR>
```

- Aside: NAME could also be given as an attribute, so in this case there is a choice between having NAME be an attribute or element.

ID's and IDREF's

- Attributes can be pointers from one object to another.
  - Compare to HTML's NAME = "foo" and HREF = "#foo".
- Allows the structure of an XML document to be a general graph, rather than just a tree.
- An ID is an identifier for an element.
  - An element can have at most 1 attribute of type ID
  - The ID value of each element in an XML document must be distinct
- An IDREF is a reference to an element ID.
- An attribute of type IDREFS contains a set of 0 or more ID values.
- However, ID's and IDREF's are untyped

Creating ID's

- In the DTD, declare, for an element *E,* an attribute *A* of type ID.
- When using tag <*E* > in an XML document, give its attribute *A* a unique value.
- Example:

  <E A = ”xyz”>

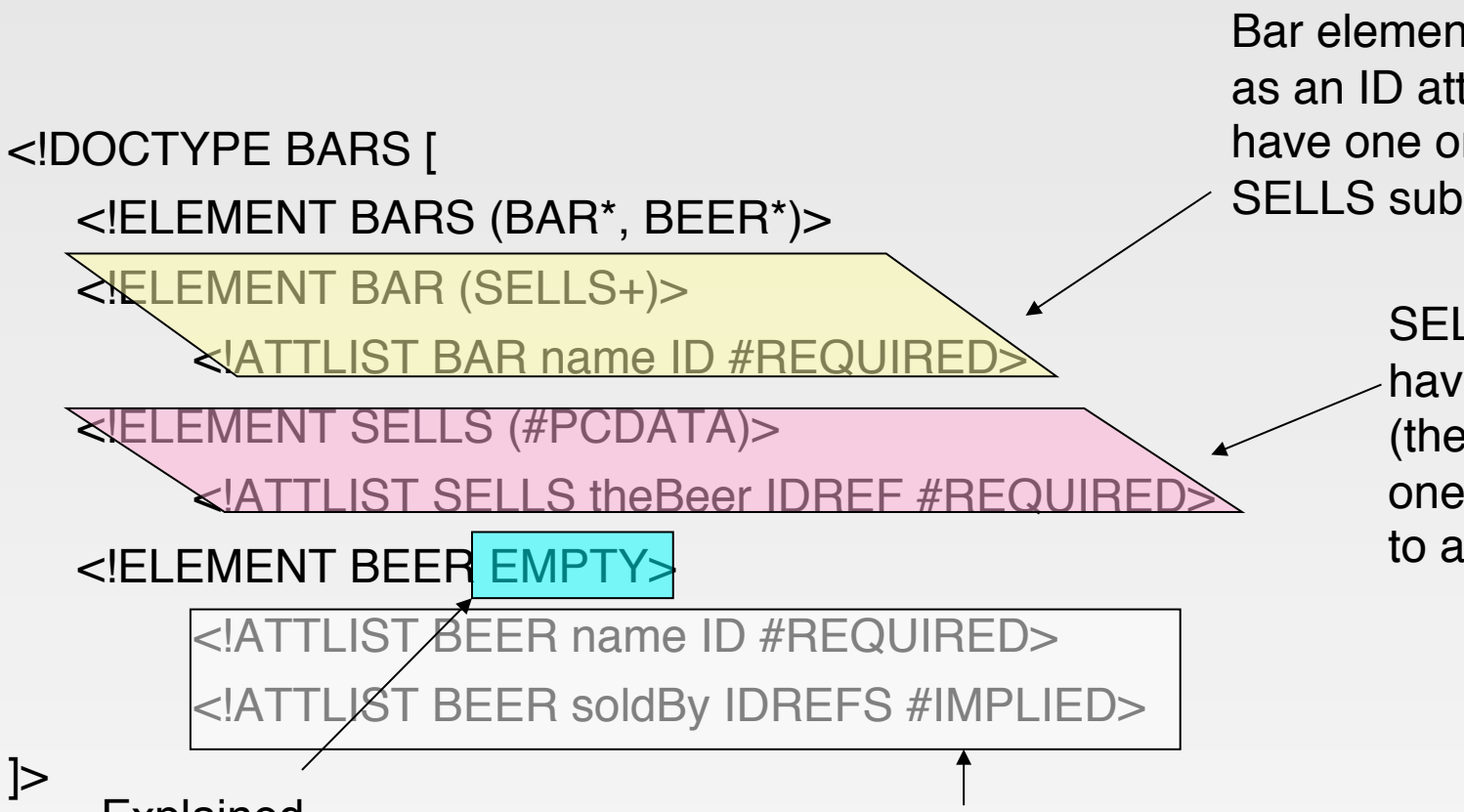
Creating IDREF's

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

Example: ID's and IDREF's

- Declare a BARS DTD to have both BAR and BEER subelements.
- BARS and BEERS have ID attribute "name".
- Element BAR has 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>
]>
Bar elements have name as an ID attribute and have one or more SELLS subelements.
SELLS elements have a number (the price) and one reference to a beer.
Explained next
Beer elements have an ID attribute called name, and a soldBy attribute that is a set of Bar names.

Example: A Document

```
<BARS>
    <BAR name = ”JoesBar”>
            <SELLS theBeer = ”G.I. Lager”>2.50</SELLS>
            <SELLS theBeer = ”Export”>3.00</SELLS>
    </BAR> …
    <BEER name = ”Export”  soldBy = ”JoesBar  SuesBar …” /> …
    </BARS>
```

Empty Elements

- We can sometimes 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:

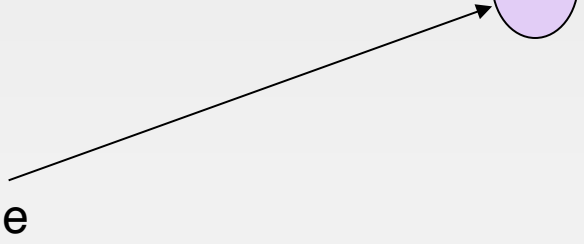
  <!ELEMENT SELLS EMPTY>

  <!ATTLIST SELLS theBeer IDREF #REQUIRED>

  <!ATTLIST SELLS price CDATA #REQUIRED>

- Example use:

  <SELLS theBeer = ”Export” price = ”2.50” />



Example: Bank DTD

```
<!DOCTYPE bank [
    <!ELEMENT bank ( ( account | customer | depositor)+)>
    <!ELEMENT account (account_number branch_name balance)>
    <!ELEMENT customer(customer_name customer_street
                                                customer_city)>
    <!ELEMENT depositor (customer_name account_number)>
    <!ELEMENT account_number (#PCDATA)>
    <!ELEMENT branch_name (#PCDATA)>
    <!ELEMENT balance(#PCDATA)>
    <!ELEMENT customer_name(#PCDATA)>
    <!ELEMENT customer_street(#PCDATA)>
    <!ELEMENT customer_city(#PCDATA)>
]>
```

Example: Bank DTD with Attributes

- Bank DTD restructured, and with ID and IDREF attribute types.

```
<!DOCTYPE bank [
    <!ELEMENT bank ( ( acct | cust )+)>
    <!ELEMENT acct (branch, bal)>
    <!ATTLIST acct
              acct_number ID          # REQUIRED
              owners      IDREFS  # REQUIRED >
    <!ELEMENT cust(cust_name, cust_st, cust_city) >
    <!ATTLIST cust
              cust_id     ID          # REQUIRED
              accts       IDREFS # REQUIRED >
    … declarations for branch, bal, cust_name,
                       cust_st and cust_city
]>
```

XML data with ID and IDREF attributes

```
<bank>
        <acct acct_number="A-401" owners="C100 C102">
                <branch_name> Downtown </branch_name>
                <bal>          500 </bal>
        </acct>
        <cust cust_id="C100" accts="A-401">
                <cust_name> Joe         </cust_name>
                <cust_st>         Monroe  </cust_st>
                <cust_city>      Madison</cust_city>
        </cust>
        <cust cust_id="C102" accts="A-401 A-402">
                <cust_name>  Mary     </cust_name>
                <cust_st>         Erin       </cust_st>
                <cust_city>     Newark </cust_city>
        </cust>
</bank>
```

DTD Limitations

- Namespaces (discussed later) are not in original design
- Very limited set of basic types – no real typing of elements and attributes

  E.g. can't constrain price to be a positive number
- Limited ways to specify data consistency constraints
  - No keys, referential integrity, no type references
- Lack of typing for ID and IDREF(S)
- Global definition of elements

XML Schema

- A more powerful way to describe the structure of XML documents.
- XML-Schema declarations are themselves XML documents.
  - I.e. in XML Schema, an XML document is described using XML
- XML Schema provides built-in types, e.g. string, integer,…
- It also allows user-defined types: restrictions of simple types, or complex types.
- Supports keys and referential integrity constraints
- Disadvantage: more complex that DTD

- Here we just give the flavour of XML Schema

Example: XML Schema for Bank

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
<xs:element name="bank" type="BankType"/>
…
<xs:element name="account">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="account_number"  type="xs:string"/>
            <xs:element name="branch_name"     type="xs:string"/>
            <xs:element name="balance"         type="xs:decimal"/>
        </xs:squence>
    </xs:complexType>
</xs:element>
….. definitions of customer and depositor ….
</xs:schema>
```

- BankType is a user-defined type, defined elsewhere in the document
- An account is defined to a be a complex type, which is a sequence.
  - The first element of the sequence is the account number, which is a string, and similarly for branch name and balance.

XML Namespaces

- There are times when XML data comes from 2 or more different sources, and so may have conflicting names
- To distinguish among different vocabularies for tags in the same document, we can use a *namespace* for a set of tags.
- To say that an element's tag should be interpreted as part of some namespace, we can use the attribute xmlns in its opening tag.
- Generally of form xmlns:*name*="URI"
- Use this for predefined XML Schema commands (tag names)

Structure of an XML-Schema Document

<? xml version = … ?>

<xs:schema xmlns:xs =”http://www.w3.org/2001/XMLschema”>

. . .

</xs:schema>

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

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

Describing Elements in XML Schema

- A key component of schemas is the *element*.
  - Use the xs:element element.
- Form of an element definition in XML Schema:

  <xs:element name = "*element name*" type = "*element type*">

  (constraints and structural information)

  </xs:element>
- 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.
     - Types are either *simple* or *complex*.

Example: xs:element

<xs:element name = ”NAME” type = ”xs:string” />

- Describes elements such as

  <NAME>Joe’s Bar</NAME>

- For a movie database:

  <xs:element name = “Title” type = “xs:string” />

  <xs:element name = “Year” type = “xs:integer” />

XML Schema: Simple Types

Simple types have no internal structure, whereas complex types do

- Compare: integer vs sequence

Simple types are one of:

- Primitive types
  - Many useful primitive types in XML Schema
    - Decimal, integer, float, Boolean, date, …
- Derived simple types using list and union constructors
- Derived simple types by restriction

Example: License Attribute for BAR

- Bars have an attribute *license*, which is one of “Full”, “Beer only” or “Beer and wine”:

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

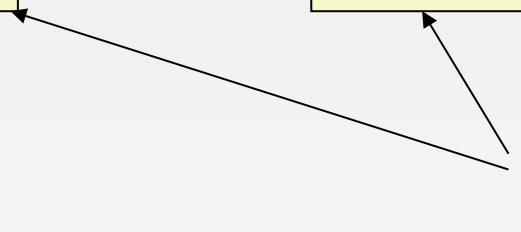
Complex Types

- To describe elements that consist of subelements, use xs:complexType.
  - Attribute name gives a name to the type, as before.
- The most common complex type is a sequence of elements.
  - The elements of a sequence are required to appear in the order given
  - The number of repetitions of an element is indefinite (unless specified otherwise)

Example: A Type for Bars

A Bar will be of type *barType*, with one subelement NAME, of which there is exactly one, followed by subelements BEERS of which there are 0 or more:

```
<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>
```



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

Keys in XML Schema

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

Keys in XML Schema

- Form:

  <xs:key name = *key name* >

  <xs:selector xpath = *path description* >

  <xs:field xpath = *path description* >

  </xs:key>

- 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 /.

Query Languages for XML

- XPath
- XQuery

The XPath/XQuery Data Model

- Corresponding to the fundamental notion “relation” of the relational model is, in XML, a *sequence of items*.
- An *item* is either:
  1. A primitive value, e.g., integer or string.
  2. A *node* (defined next).
- The main node types are:

1. *Document nodes* are files containing an XML document (perhaps denoted by a local path or URL).
2. *Elements* are pieces of a document consisting of some opening tag, its matching closing tag (if any), and everything in between.
3. *Attributes* are names that are given values inside opening tags.

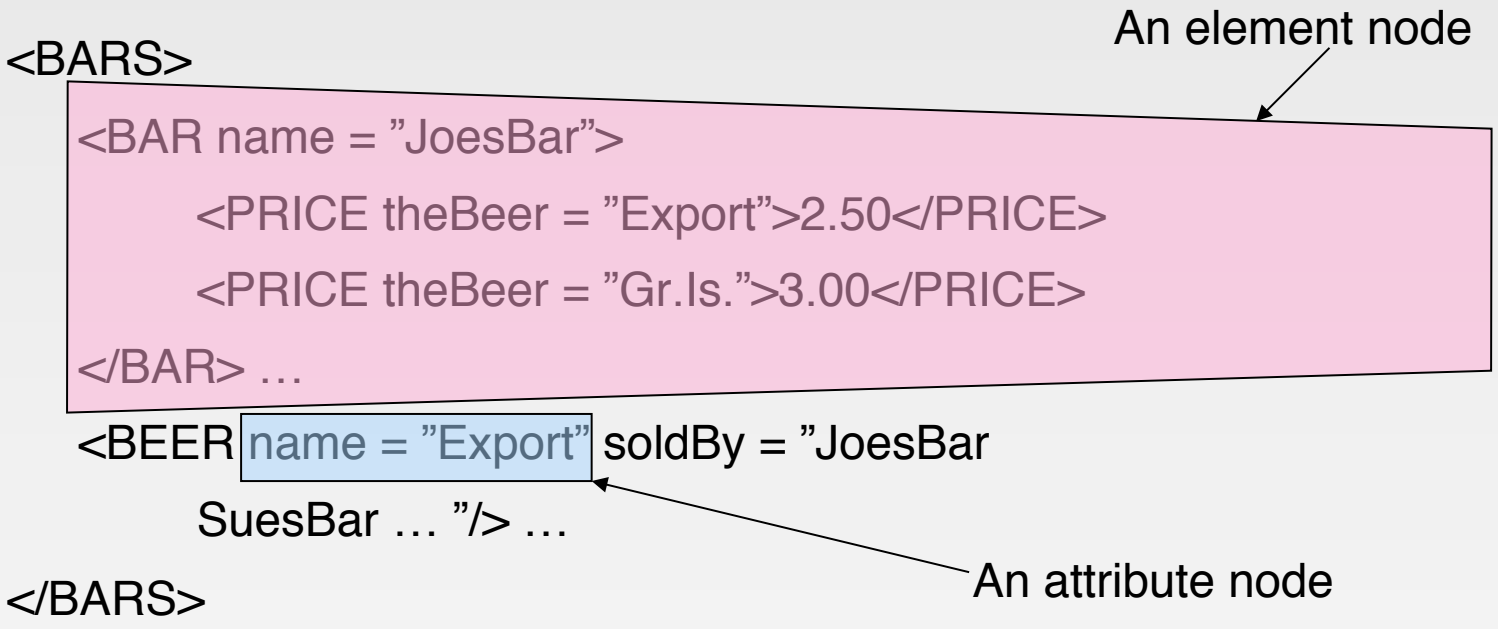
Document Nodes

- Formed by doc(URL) or document(URL) (or doc(filename) or document (filename)) .
- 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

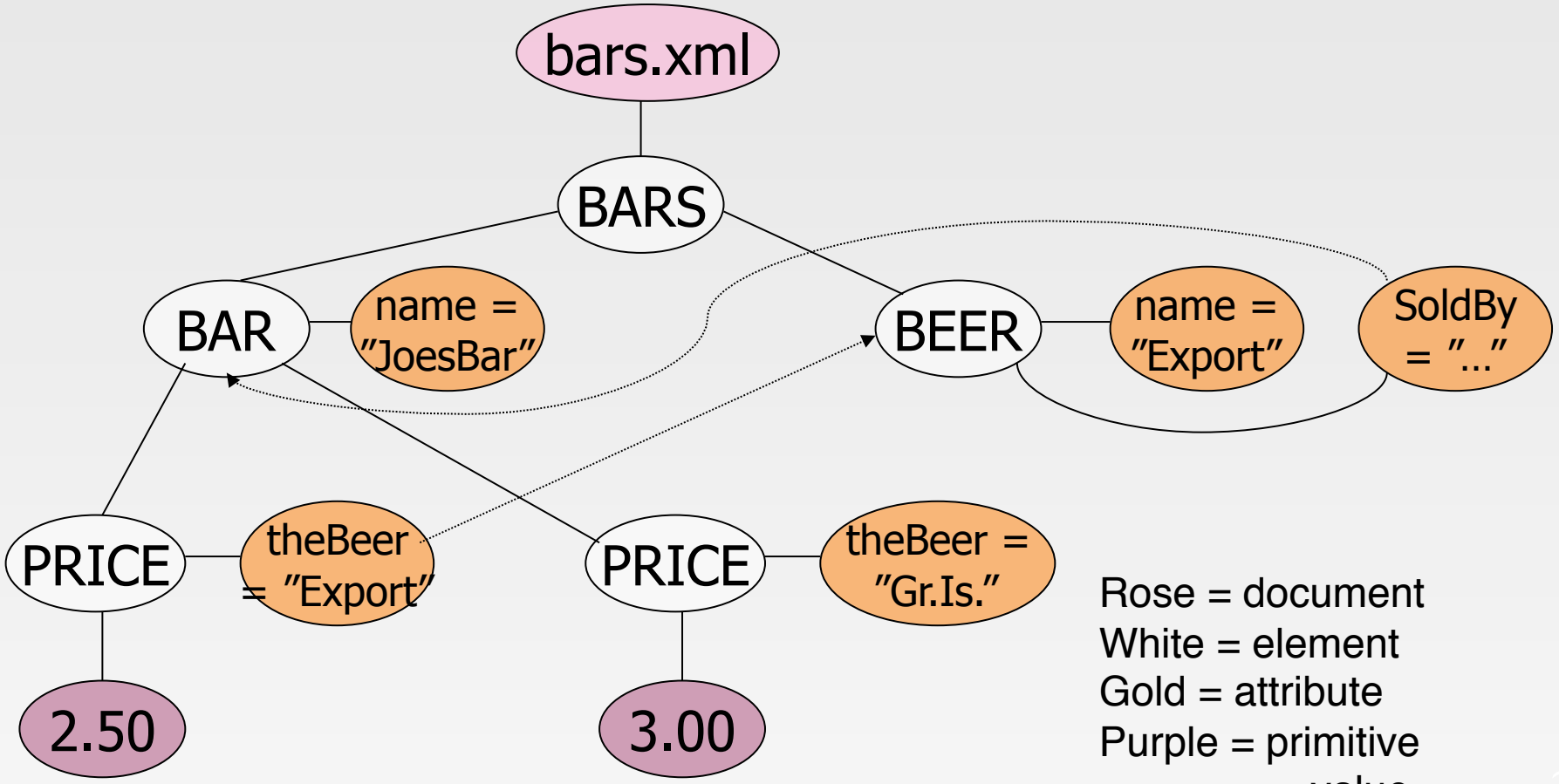
```
<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*, BEER*)>
    <!ELEMENT BAR (PRICE+)>
        <!ATTLIST BAR name ID #REQUIRED>
    <!ELEMENT PRICE (#PCDATA)>
        <!ATTLIST PRICE theBeer IDREF #REQUIRED>
    <!ELEMENT BEER EMPTY>
        <!ATTLIST BEER name ID #REQUIRED>
        <!ATTLIST BEER soldBy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

Example Document



The document node is all of this, plus the header <? xml version… ?>.

Nodes as Semistructured Data



Paths in XML Documents

- XPath is a language for describing paths in XML documents.
- The result of the described paths is a sequence of items.
- Compare with SQL:
  - SQL is a language for describing relations in terms of other relations.
  - The result of a query is a relation (bag) made up of tuples.

Path Expressions

- Simple path expressions are sequences of slashes (/) and tags, starting with /.
  - Example: /BARS/BAR/PRICE
- The format used is strongly reminiscent of UNIX naming conventions.
- Construct the result by starting with the doc node and processing each tag from the left.

Evaluating a Path Expression

Recursive evaluation:

- 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 = ”Export”>2.50</PRICE>
        <PRICE theBeer = ”Gr.Is.”>3.00</PRICE>
    </BAR> …
    <BEER name = ”Export” soldBy = ”JoesBar
        SuesBar … ”/> …
</BARS>
```

One item, the BARS element

Example: /BARS/BAR

```
<BARS>
    <BAR name = ”JoesBar”>
        <PRICE theBeer =”Export”>2.50</PRICE>
        <PRICE theBeer = ”Gr.Is.”>3.00</PRICE>
    </BAR> …
    <BEER name = ”Export” soldBy = ”JoesBar
        SuesBar …”/> …
</BARS>
```


This BAR element followed by all the other BAR elements

Example: /BARS/BAR/PRICE

<BARS>

<BAR name = ”JoesBar”>

<PRICE theBeer =”Export”>2.50</PRICE>

<PRICE theBeer = ”Gr.Is.”>3.00</PRICE>

</BAR> …

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

</BARS>

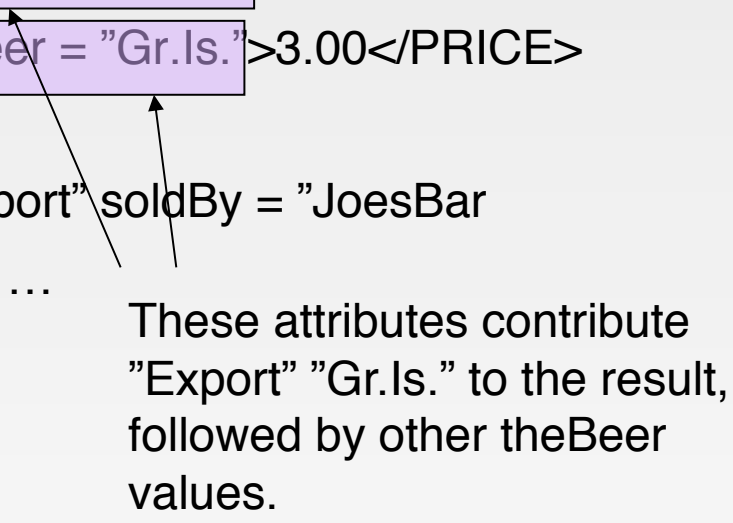
These PRICE elements followed by the PRICE elements of all the other bars.

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 = ”Export”>2.50</PRICE>
        <PRICE theBeer = ”Gr.Is.”>3.00</PRICE>
    </BAR> …
    <BEER name = ”Export” soldBy = ”JoesBar
        SuesBar …”/> …
</BARS>
```



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*.
- In general, *//X* finds all *X* subelements at the “current” position or in descendants.

Example: //PRICE

```
<BARS>
    <BAR name = ”JoesBar”>
        <PRICE theBeer =”Export”>2.50</PRICE>
        <PRICE theBeer = ”Gr.Is.”>3.00</PRICE>
    </BAR> …
    <BEER name = ”Export” soldBy = ”JoesBar
        SuesBar …”/> …
</BARS>
```

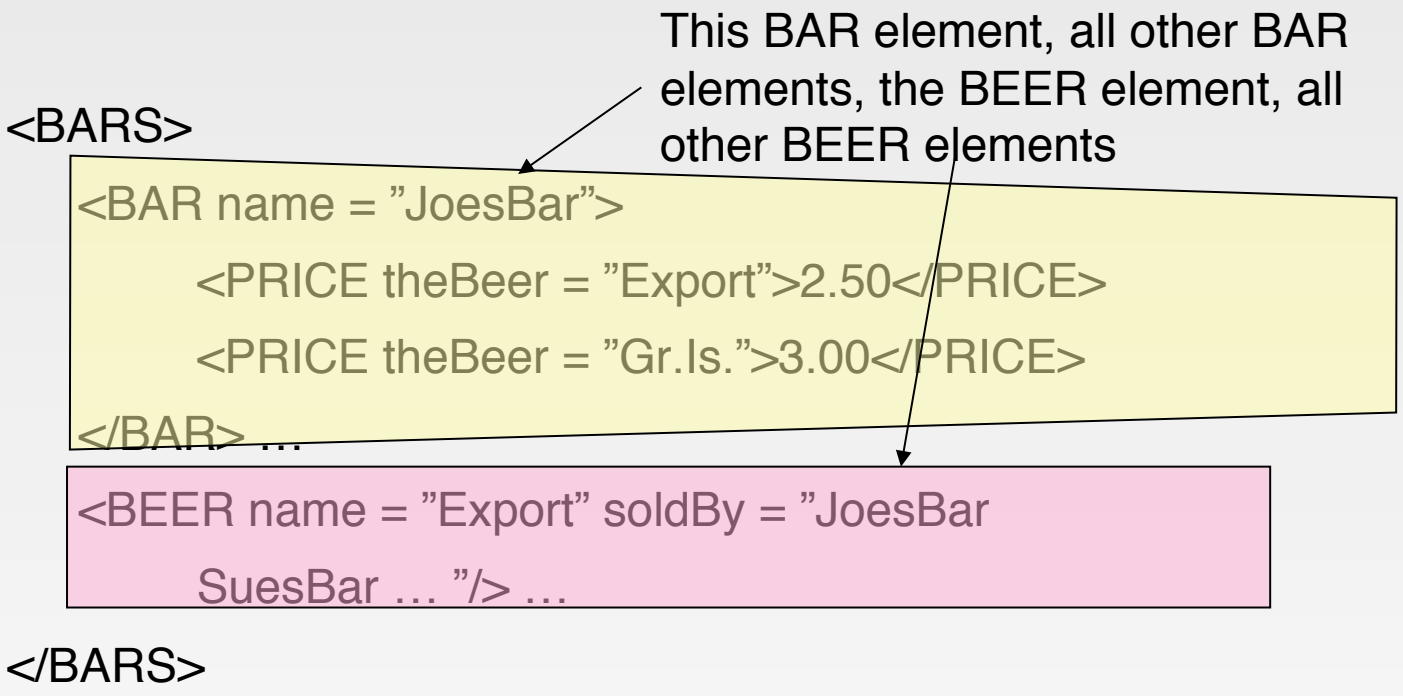
These PRICE elements and any other PRICE elements in the entire document

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.
- Likewise @* means “any attribute”

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.
- Values can be compared with operators such as <, <=, =, !=, etc.

Example: Selection Condition

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

<BARS>

<BAR name = ”JoesBar”>

<PRICE theBeer = ”Export”>2.50</PRICE>

<PRICE theBeer = ”Gr.Is.”>3.00</PRICE>

</BAR> …

The current element.

The condition that the PRICE be < $2.75 makes this price, but not the Gr.Is. price, part of the result.

Example: Attribute in Selection

- /BARS/BAR/PRICE[@theBeer = ”Gr.Is.”]

<BARS>

<BAR name = ”JoesBar”>

<PRICE theBeer = ”Export”>2.50</PRICE>

<PRICE theBeer = ”Gr.Is.”>3.00</PRICE>

</BAR> …

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

Another Example

- From the text:

  /StarMovieData/Star[.//City=“Malibu]/Name

  - Note that “City” isn’t part of the path expression to Name.
  - Note the “.” before “//” (the text has this wrong)

Example:

… <Star starID = “hf”>

<Name>Harrison Ford</Name>

<City>Malibu</City>

</Star>

Other Forms of Conditions

- A tag by itself, [T], is true only for elements that have 1+ subelements with tag T.
  - Can combine with [T1 or T2] or [T1 and T2].
  - E.g. /BARS/BAR/PRICE[@theBeer = ”Gr.Is.” or @theBeer = “Ex”]
- An attribute, [A], by itself is true for elements that have a value ofr the attribute A.

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*, or modes of navigation.
- The default axis is child:: --- go to all the children of the current set of nodes.

Example: Axes

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

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. also: ancestor, ancestor-or-self, next-sibling, previous-sibling, 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 as XPath.
- XQuery is an expression language.
  - Like relational algebra --- any XQuery expression can be an argument of any other XQuery expression.

Item Sequences

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

Empty
sequence

FLWR Expressions

- The most important form of XQuery expressions involves for-, let-, where-, return- (FLWR) clauses.

1. A query begins with one or more for and/or let clauses.
   - The for's and let's can be interspersed.
2. Then an optional where clause.
3. A single return clause.

Form:

for *variable* in *expression*

let *variable* := *expression*

where *condition*

return *expression*

Semantics of FLWR Expressions

- Each for creates a loop.
- A let produces a local variable assignment.
- 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.
  - So return can be thought of as “add to result”

FOR Clauses

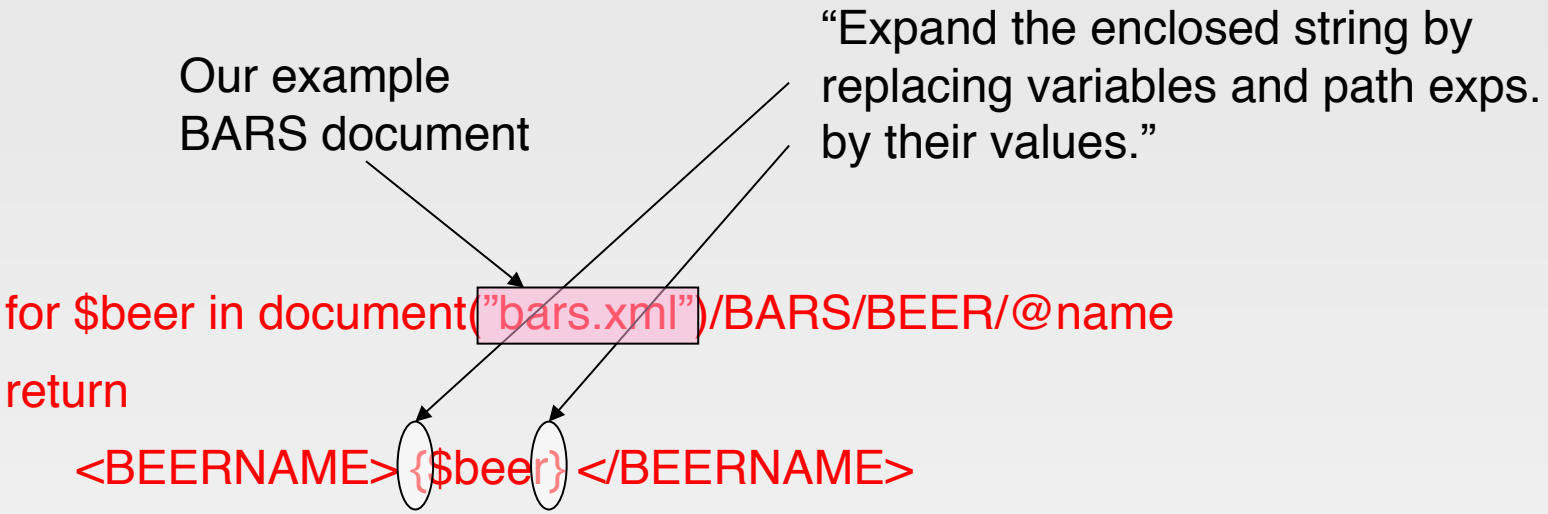
Form:

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.
- E.g.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
        … (do something with each Movie element)
```

Example: FOR



- $beer ranges over the name attributes of all beers in our example document.
- The result is a sequence of BEERNAME elements:
  <BEERNAME>Export</BEERNAME> <BEERNAME>Gr.Is.
  </BEERNAME> . . .
- N.B.: Text inside tags is surrounded by {…} in order to be interpreted as a XQuery expression

Use of Braces

- When a variable name like $x, or an expression, could be text, we need to surround it by braces to avoid having it interpreted literally.
- This happens most often between tags or for values of attributes, where any text string is permissable
- Example:
  - <A>$x</A> is an A-element with value ”$x”, just as <A>foo</A> is an A-element with ”foo” as value.
  - <A>{$x}</A> evaluates $x and puts its value between the tags.

Use of Braces --- (2)

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

  return “$x”.

LET Clauses

Form:

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>Export Gr.Is. …</BEERNAMES>

  (Q: What if the second *let* is replaced by *for*?)

Order-By Clauses

- FLWR is really FLWOR: an order-by clause can precede the return.
- Form:

  order by <expression>
- Can have 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 Export, lowest price first.

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

Recall the DTD:

```
<!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: Comparisons

- Let us produce the PRICE elements (from all bars) for all the beers that are sold by Joe's Bar.
  - Note that we're essentially doing a join on bars and beers.
- The output will be elements with tag BBP and 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 $beer/@name = $price/@theBeer
return <BBP bar = {$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=”Export”] eq ”2.50”

  is true if Joe charges $2.50 for Export.

Comparison of Two Elements

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

  /BARS/BAR[@name="JoesBar"]/PRICE[@theBeer="Export"]

  eq

  /BARS/BAR[@name="SuesBar"]/PRICE[@theBeer="Export"]

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

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

- Various types of expressions can be interpreted as true or false when used in a condittion.
- 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
     1. 0,
     2. “” [the empty string], or
     3. () [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 E_1, producing a sequence.
2. Let $x 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
```

Note: 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
```

Set Operators

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

Aggregations

- XQuery allows the usual aggregations, such as sum, count, max, min.
- They take any sequence as argument.
- E.g. find bars where all beers are under $5.

```
let $bars = doc("bars.xml")/BARS
for $price in $bars/BAR/PRICE
where max($price) < 5
return $bar/BAR/@name
```

End: XML

Document Order

- Comparison by document order: << and >>.
- Example:

  $d/BARS/BEER[@name="Export"] << $d/BARS/BEER[@name="Gr.Is."]

  is true iff the Export element appears before the Gr.Is. element in the document $d.