

DATALOG: Logic-based Information Management

Datalog₀: IM based on subset of FO logic

Example: want to manage information about family relationships

- *Tell facts* like “parents of chuck are liz and phil; chuck is male;”
- *Ask questions* like “Is chuck male? Is phil a father? Who is chuck’s mother? Who are all the males?”

Use notation of FOL for both languages.

- *predicates/relations:* male, parents
- *constants:* ed, vicky, ... *(must start with lowercase)*
- *ground atomic formula:* $\langle \text{pred} \rangle (\langle \text{constt} \rangle , \dots) .$
- $\mathcal{L}_{\text{tell}}$ = ground atomic formulas

```
male(phil).      female(liz).  
male(chuck).    female(di).  
  
parents(chuck,liz,phil).  
parents(ann,liz,phil).  
parents(andy,di,chuck).  
%% parents(child,mother,father)
```

```
male(phil).    female(liz).  
male(chuck).   female(di).  

```

```
%% parents(child,mother,father)  
parents(chuck,liz,phil).  
parents(ann,liz,phil).  
parents(andy,di,chuck).  

```

$\mathcal{L}_{question} = \mathcal{L}_{tell}$
 $\mathcal{L}_{answer} = \{yes, no\}$

We can (sort of) use the Prolog programming language as a Datalog IM implementation.

```
?- male(phil).           yes  
?- female(chuck).        no  
?- male(andy).           no  
?- parents(chuck,liz,phil). yes  
?- parents(liz,chuck,phil).  

```

?

Datalog_{0.25}: queries with variables

- **variables:** **Y, A, Who** (by convention, start with caps)
 - **argument:** *constant or variable*
 - **atomic formula:** *<predicate> (<argument> , ...) .*
- $\mathcal{L}_{question}$ = atomic formulas (may have variable)
- \mathcal{L}_{answer} = {yes,no} or variable substitutions

```
likes(eve,pie).  person(tom).  
likes(al,eve).  food(pie).  
likes(eve,tom). food(apple).  
likes(eve,eve).
```

Datalog_{0.25}: queries with variables

- **variables:** **Y, A, Who** (*by convention, start with caps*)
- **argument:** *constant or variable*
- **atomic formula:** *<predicate> (<argument> , ...) .*

$\mathcal{L}_{question}$ = atomic formulas (may have variable)

\mathcal{L}_{qnsver} = {yes,no} or variable substitutions

```
likes(eve,pie).  person(tom).  
likes(al,eve).   food(pie).  
likes(eve,tom).  food(apple).  
likes(eve,eve).
```

```
?-likes(al,Who).  
    Who=eve  
?-likes(eve,W).  
    W=pie   ;  
    W=tom   ;  
    W=eve  
?-likes(A,B).  
    A=eve,B=pie ;  
    A=al,B=eve  ;  
    ...  
?-likes(A,A)  
    D=eve
```

(Aside: Datalog is more “elegant/succinct” than SQL)

Datalog:

```
likes(eve,pie).  person(tom).  
likes(al,eve).  food(pie).  
likes(eve,tom). food(apple).  
likes(eve,eve).
```

```
?-likes(al,Whom).
```

Corresponding SQL:

```
CREATE TABLE likes(who VARCHAR; whom VARCHAR;  
                    PRIMARY KEY(who,whom));  
INSERT INTO likes VALUES ('eve','pie');  
...
```

```
SELECT t.whom  
FROM likes t  
WHERE t.who='al';
```

Datalog_{0.5}: conjunctive queries

```
likes(eve,pie).  person(tom).  
likes(al,eve).   food(pie).  
likes(eve,tom).  food(apple).  
likes(eve,eve).
```

$\mathcal{L}_{question}$ = conjunction of atomic formulas
'and' is written as comma ',' in Prolog

```
?-likes(eve,W) , person(W).  
    W=tom  
?-likes(eve,V) , likes(al,V).  
    V=eve  
?-likes(eve,W) , likes(al,V) , V=W.  
    V=eve  
?-likes(eve,W) , person(W) , food(V)  
    W=tom, V=pie ;  
    W=tom, V=apple  
?-likes(eve,W) , likes(W,V).  
    W=eve, V=pie ;  
    W=eve, V=tom ;  
    W=eve, V=eve
```

Datalog RULES

a) Rules as shorthand for some queries

```
likes(eve,pie).  person(tom).  
likes(al,eve).   food(pie).  
likes(eve,tom).  food(apple)  
likes(eve,eve).
```

```
q1 :- likes(eve,V), person(V).
```

```
?-q1.  
yes
```

“Is there someone whom Eve likes?” (hides uninteresting vars)

```
q2(Who) :- likes(Who,F), food(F).
```

```
?-q2(H).  
H=eve
```

*“Who likes some food?”
(hides some variables not of interest -- the food F liked by that person)*

b) Rules for error checking

```
likes(eve,pie).      person(tom).  
likes(al,eve).       food(pie).  
likes(eve,tom).      food(apple)  
likes(eve,eve).
```

```
error :- likes(X,_),food(X).  
error :- food(Y),person(Y).
```

?-error.

*These are not
queries but rules
added to the “program”!*

*Note possibility of multiple rules defining the same idea.
Return YES if any say YES. Return NO if all fail.*

c) Rules for defining new predicates

```
male(phil).    female(liz).  
male(chuck).   female(di).  
  
parents(chuck,liz,phil).  
parents(ann,liz,phil).  
parents(andy,di,chuck).  
%% parents(child,mother,father)
```

```
sibling(X,Y):-parents(X,M,F),parents(Y,M,F),NOT(X=Y).
```

A rule has

- **head** -- atom (sometime called **goal**)
- **body** -- conjunctive query (atoms sometimes called **subgoals**)
- can be interpreted naturally as “*conclude head if body*”, with variables not appearing in the head quantified as “there exists” at the left end of the body

“sibling(X,Y) if exist M,F such that parents(X,M,F),and
parents(Y,M,F) and NOT (X=Y)”

EXAMPLES

Compute grandparents:

e is the parent of d, e is the parent of f,
d is the parent of c, f is the parent of g,
f is the parent of h, h is the parent of i,
h is the parent of j, j is the parent of k,
e is the parent of d, d is the parent of c,
c is the parent of b, b is the parent of a.

```
grandparent(X,Y) :- parent(X,Z) , parent(Z,Y)
```

RECURSIVE

Compute ancestors:

```
ancestor(X,Y) :- ancestor(Z,Y) , parent(X,Z)  
ancestor(X,Y) :- parent(X,Y)
```

SEMI-STRUCTURED DATABASES

Overview of MongoDB

JavaScript Syntax:

Two ways of writing a javascript file:

*** Inserted in a web page (html)**

```
<html>
  <body>
    <script language="javascript" type="text/javascript">
      ..... the code goes here
    </script>
  </body>
</html>
```

*** Standalone file with extension .js**

```
<head>
<script type='text/javascript' src='file1.js'>
</script>
```

Simple output to web pages

- Write html code to current page:
`document.write("hello
");`

Variables:

- Variables must be declared:
`var x;`
- The type is assigned dynamically based on its value:
`x=3; // type will be integer`
- Control structures as in Java:
`for (i=0;i<20;i++){ }
if (i<1){ } else { }`
- Assignment and conditional together:
`var result = x<3 ? "less" : "greater";`

Strings:

- Declaring as a variable
var x = "Hello";
- Declaring a string as an object:
var y = new String("Hello");
- Some string attributes and methods:
length (is an attribute)
- **Functions** without types:
 - function add(x,y){
 var z=x+y;
 return z;
}
- Function call:
var w=add(3,4);

JSON (Javascript Serializable Object Notation)

We view this as another notation for representing and storing information. It is relevant because

- It is relatively easy to translate objects in OO languages into it*
- It is being used as a more light-weight XML*
- A number of NoSQL database systems (e.g., MongoDB), which call themselves "document stores", are based on JSON as a data model.*

Serialization of an Object:

Objects (excluding methods) can be converted into a string of bytes that can be stored in a file or transmitted.

This process is also called Marshalling.

JSON syntax

A JS is defined recursively

a JS is either

* a **_value_** :

number e.g. 25

string e.g. "firstName"

boolean

NULL keyword

* an **_array_** (ordered list of JSes indexed by integers from 0) of JS,
enclosed by [and], with elements separated by commas. e.g.
["skiing", "arguing", "sleeping"]

* an **_object_** (associative array -- unordered list of (string:JS) pairs
enclosed between { and }

e.g.

```
{  
  "type": "home",  
  "number": "212 555-1234"  
},
```

Example of a JSON document

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "hobbies" : ["skiing", "arguing", "sleeping"]
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

As the example above illustrates, arrays and objects can nest to arbitrary depth.

At this point there are no widely accepted standard notations for imposing "schemas" on JSON objects, or for querying them.

MongoDb uses JSON documents to build collections.

MongoDb is based on Javascript.

Hierarchy:

MongoDb:

Database ---> Collections ---> Documents ---> Fields

Relational (SQL) Db:

Database ---> Tables ---> Tuples ---> Columns

The fields are defined at the Documents level instead of at the Collections level

USES javascript

Object use JSON (Javascript Object Notation)

BSON (internally binary)

MongoDb:

You can execute a version on a server:

```
mongod -dbpath "C:\Users\Antonio Miranda\Desktop\cs539\data\db"
```

You can connect using command line interface

```
mongo
```

or create a connection from an application:

```
myConn=new Mongo("localhost");  
myDb=myConn.getDB("examples"); // database name
```

Or both at the same time:

```
myDb=connect("localhost/examples");
```

Other commands (similar to mySQL):

Show databases

use _____

show collections

db.dropDatabase()

db._____.drop()

Finding Collections in a database:

```
db.getCollectionNames()
```

Inserting an document into a collection:

```
db.students.insert(object)
```

Example 1:

Create the following document:

Name: Smith
Address: 256 Main St
City: Piscataway
State: NJ
Age: 25
Married: false,
Gender: male

Delete the “students” collection
and insert the new document into the “students” collection.

The “students” collection should have only one document at this point.

Example 1:

```
var student={
    name:"Smith",
    address:"256 Main St",
    city:"Piscataway",
    state:"NJ",
    age:25,
    married:false,
    gender:"male"
};

print (student.name);

myConn=new Mongo("localhost");
myDb=myConn.getDB("examples");

myDb.students.drop();
myDb.students.insert(student);
```

Examples can be executed using:

```
mongo ex01.js
```

Inserting several documents with one command:

Insert also can insert several documents, just store them in an array:

Example 2:

```
unicorn_array=[
  {name: 'Happy',
    dob: new Date(1992,2,13,7,47),
    loves: ['carrot','papaya'],
    weight: 600,
    gender: 'm',
    vampires: 63},
  {name: 'Aurora',
    dob: new Date(1991, 0, 24, 13, 0),
    loves: ['carrot', 'grape'],
    weight: 450,
    gender: 'f',
    Vampires: 43},
  ...
  ...

];
db=connect("localhost/examples");
db.unicorns.drop();
db.createCollection("unicorns",{capped:false});
db.unicorns.insert(unicorn_array);
```

(see ex02.js)

Query using the find method, and cursors:

```
db.unicorns.find()    // returns the list of all documents
```

Notice that we can define a cursor (like a result set):

```
cursor=db.unicorns.find()
```

Two different ways:

1) Using a normal loop on the cursor

```
while (cursor.hasNext()) {  
    var u=cursor.next();  
    print(u.name);  
}
```

2) Using the cursor's forEach method

```
cursor.forEach(function(e) {print(e.name);});
```

(see ex03.js)

Select the fields to return.

Notice that the ID field by default is displayed unless it is set to false.

Example:

Return only the name of each unicorn

```
cursor=db.unicorns.find({}, {name:true, _id:false});  
  
while (cursor.hasNext()) {  
    var u=cursor.next();  
    printjson(u);  
}
```

(see ex04.js)

Example (using functions in javascript):

Display (nicely formatted) the weight of each unicorn in Kg.

The conversion from pounds is: $\text{Kg} = \text{pounds} * 0.454$

```
function padRight(st,n) {
    if (st.length<n) {
        for (var i=0;i<n-st.length;i++)
            st=st+' ';
    }
    return st;
}

var cursor=db.unicorns.find({});
cursor.forEach(
    function(e) {
        var wkg=e.weight*0.454;
        print(padRight(e.name,30)+'\t'
            weight='+wkg.toFixed(2)+'Kg'
        );
    }
);
```

(see ex05.js)

Using conditions:

`$lt, $gt, $gte, $lte, $eq, $ne`

Example: Find all unicorns with weight greater than 700 pounds:

```
db.unicorns.find({weight:{$gt:700}})
```

Other options in find:

```
db.collection.find({field:{$exists:false}}) // returns all
                                              documents without
                                              field
{$in:[xxx,xxx,xxx,xxx]} // if a field is one of them
$or:[{field1:value1},{field2:value2}]
```

Example: Unicorns who love grapes or weigh more than 700 pounds

```
db.unicorns.find({$or:[{loves:'grape'},{weight:{$gt:700}}]})
```

Update:

```
db.collection.update({}, { use update operators })
```

Update operators:

```
$set, $inc, $min, $max, $mul, $rename  
$setOnInsert, $unset, $push
```

`$min` updates only if new value is smaller than current.

`$max` updates only if new value is larger than current.

`$rename: { "a": "b", "c": "d" }` (original field name and new field name)

Adding the `{upsert:true}` option inserts if not found
(like on duplicate key update)

Adding the `{multi:true}` option will allow to update several pages.

Example:

```
db.unicorns.update({name:'Dunx'}, {$set:{weight:100}})
```

With upsert:

```
db.unicorns.update({name:'Nice'},  
    {$set:{weight:804,gender:'m'}}, {upsert:true})
```

XML and Semi-structured data

What is XML?

- Text annotation/markup language (“eXtensible Markup Language”)

```
<BOOK genre="Science" format="Hardcover">  
  <AUTHOR>  
    <FIRSTNAME>Rich</FIRSTNAME>  
    <LASTNAME>Feynman</LASTNAME>  
  </AUTHOR>  
  <TITLE>The Character of Physical Law</TITLE>  
  <PUBLISHED>1980</PUBLISHED>  
</BOOK>
```

- Think of markup as *meta-data* (*data about data*)
- Resulting document is structured like a tree

<h1> Bibliography </h1>

<p> <i> Foundations of Databases </i>

Abiteboul, Hull, Vianu

 Addison Wesley, 1995

<p> <i> Data on the Web </i>

Abiteoul, Buneman, Suciua

 Morgan Kaufmann, 1999

<bibliography>

<book> <title> Foundations... </title>

<author> Abiteboul </author>

<author> Hull </author>

<author> Vianu </author>

<publisher> Addison Wesley </publisher>

<year> 1995 </year>

</book>

...

</bibliography>

HTML vs XML

- Syntax like html, but set of tags is not fixed
- Separate web page *content* (specified in XML) from *display format* (specified in different language, XSL); plus language(s) for *transforming* document structure (XSLT, XQuery)

Success of XML

- Ability to represent “varying format data” (semi-structured)
- Ability to introduce new tags, led to publication of “standards” for many sub-areas.

data exchange

- Example: Chemical Markup Language

<molecule>

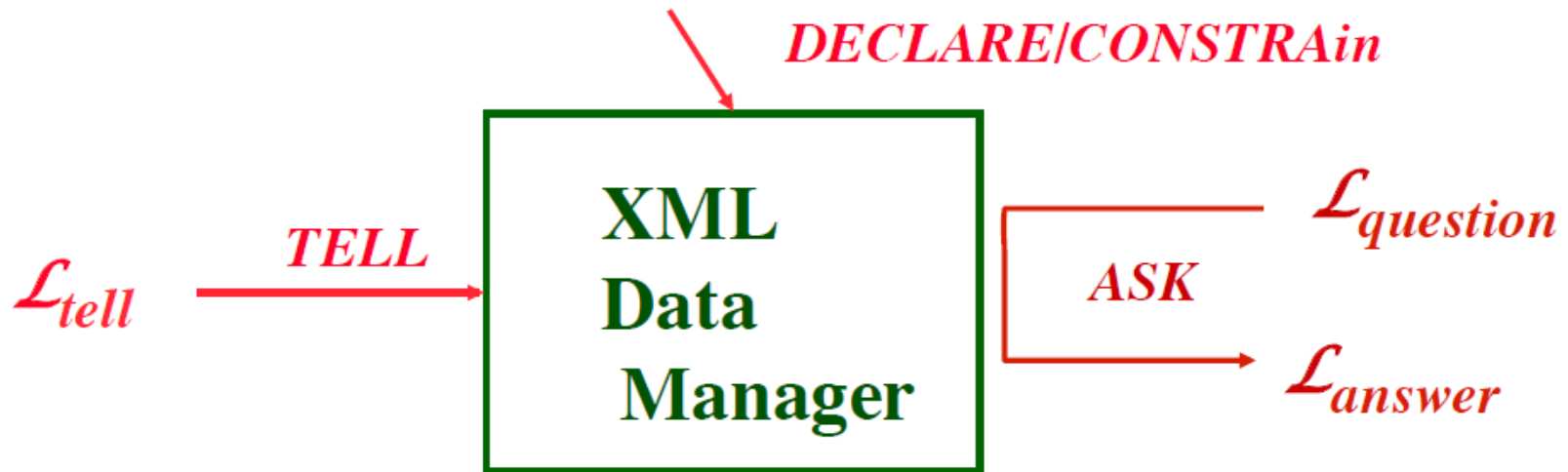
<weight>234.5</weight>

<Spectra>...</Spectra>

<Figures>...</Figures>

</molecule>

Semi-structured Data Management



$\mathcal{L}_{tell} = \text{XML document}$

$\mathcal{L}_{question} = \text{XPath, (XQuery)}$

$\mathcal{L}_{answer} = \text{XML document}$

$\mathcal{L}_{declare} = \text{DTD (, XML Schema)}$

XML Terminology

- tags: **book**, **title**, **author**, ...
- start tag: **<book>**, end tag: **</book>**
- elements:
 - <book>...</book>**
 - <author>...</author>**
- elements are nested and bottom out at data values (hence form a tree)
- empty element: **<red></red>** abbrev. **<red/>**
- an XML document must be a *single element* (“*root*”)

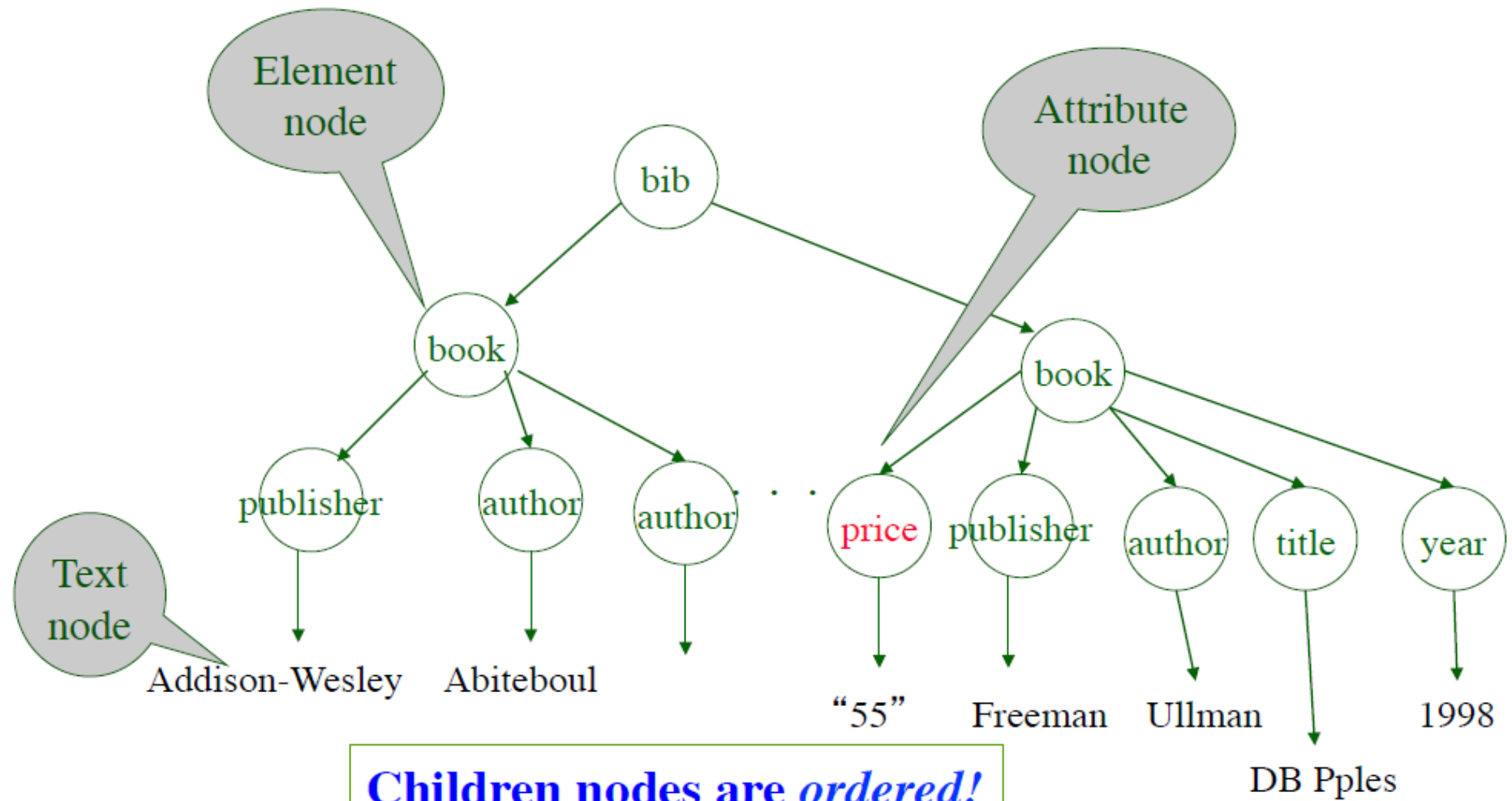
(a well formed XML document has properly nested matching tags)

Example doc for XPath Queries

```
<bib>
  <book> <publisher> Addison-Wesley </publisher>
    <author> Serge Abiteboul </author>
    <author> <first-name> Rick </first-name>
      <last-name> Hull </last-name>
    </author>
    <author> Victor Vianu </author>
    <title> Foundations of Databases </title>
    <year> 1995 </year>
  </book>
  <book price="55">
    <publisher> Freeman </publisher>
    <author> Jeffrey D. Ullman </author>
    <title> Principles of Database and Knowledge Base Systems </title>
    <year> 1998 </year>
  </book>
</bib>
```

The ordered tree view of an XML document

(draw on board)

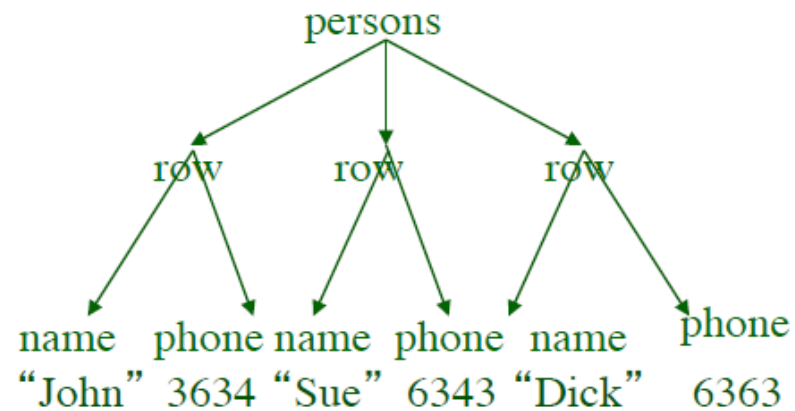


One way to represent Relational Data in XML

XML:

persons

name	phone
John	3634
Sue	6343
Dick	6363



```
<persons>
  <row> <name>John</name>
    <phone> 3634</phone></row>
  <row> <name>Sue</name>
    <phone> 6343</phone>
  <row> <name>Dick</name>
    <phone> 6363</phone></row>
</persons>
```

XML vs Relational Data

- XML is self-describing
- schema elements become part of the data (tags):
person(name,phone) vs
<person> <name> </name> <phone> </phone> ...
- so XML is more flexible because do not have to follow slavishly a single flat schema:

SEMISTRUCTURED DATA

Semistructured Data

- fields may be **missing**

```
<person> <name>bob</name> <phone>5-4544</phone></person>
```

```
<person> <name>anna</name> </person>
```

- fields may be **repeated**

```
<person> <name>bob</name>
```

```
<phone>5-4544</phone>
```

```
<phone> 3-5436</phone> </person>
```

- fields may be **nested**

```
<person>
```

```
<name> <first>bob</first><last>jones</last></name>
```

```
</person>
```

- fields may be **heterogeneous**

```
<name> <first>bob</first><last>jones</last></name>
```

```
<name> <first>bob</first><mid> t</mid> <last>jones</last></name>
```

- collections** may be **heterogeneous**

```
<persons>
```

```
<teacher> ... </teacher>
```

```
<student> ...</student>
```


```
</persons>
```


DTD – Document Type Definition


DTD – An Example

```
<!ELEMENT Basket (Cherry+, (Apple | Orange)*) >
<!ELEMENT Cherry EMPTY>
  <!ATTLIST Cherry flavor CDATA #REQUIRED>
<!ELEMENT Apple EMPTY>
  <!ATTLIST Apple color CDATA #REQUIRED>
<!ELEMENT Orange EMPTY>
  <!ATTLIST Orange location 'Florida' >
```

2 documents:



```
<Basket>
  <Cherry flavor= 'good' />
  <Apple color= 'red' />
  <Apple color= 'green' />
</Basket>
```



```
<Basket>
  <Apple/>
  <Cherry flavor= 'good' />
  <Orange/>
</Basket>
```

** DTDs as Grammars*

- A DTD = a grammar
- A valid XML document = a parse tree for that grammar

** Problem with DTD: not in XML notation!!!*

There is more advanced notation: XML Schema (not in this course!)

Querying XML Documents

Xpath: Summary

<code>bib</code>	matches a bib element
<code>*</code>	matches any element
<code>/</code>	matches the root element
<code>/bib</code>	matches a bib element under root
<code>bib/paper</code>	matches a paper in bib
<code>bib//paper</code>	matches a paper in bib, at any depth
<code>//paper</code>	matches a paper at any depth
<code>//paper/..</code>	matches the parent of paper at any depth
<code>paper book</code>	matches a paper <i>or</i> a book
<code>@price</code>	matches a <code>price</code> attribute
<code>bib/book/@price</code>	matches <code>price</code> attribute in book, in bib
<code>bib/book/[@price<"55"]/author/lastname</code>	matches...