XML and Databases

Lecture 1
Introduction to XML

Sebastian Maneth
NICTA and UNSW

CSE@UNSW - Semester 1, 2010

XML and Databases

Lecture 1
Introduction to XML

Sebastian Maneth
NICTA and UNSW
(today: Kim Nguyen)

CSE@UNSW - Semester 1, 2010

XML

→ Similar to HTML (Berners-Lee, CERN → W3C)
use your own tags.

→ Amount/popularity of XML data is growing steadily
(faster than computing power)

XML

→ Similar to HTML (Berners-Lee, CERN → W3C)
use your own tags.

→ Amount/popularity of XML data is growing steadily
(faster than computing power)

HTML pages are *tiny* (couple of Kbytes)

XML documents can be **huge** (GBytes)

↓

Databases

XML and Databases

This course will

→ introduce you to the world of XML and to
the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data.

XML and Databases

This course will

→ introduce you to the world of XML and to
the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data.

can handle huge amounts of data stored in **relations**
→ storage management
→ index structures
→ join/sort algorithms
→ …

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data

tree structured

can handle huge amounts of data stored in **relations**
→ storage management
→ index structures
→ join/sort algorithms
→ …

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

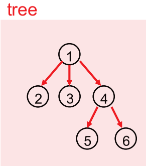
Some of these challenges are

Existing (DB) technology can**not** be applied to XML data

tree structured

can handle huge amounts of data stored in **relations**
→ storage management
→ index structures
→ join/sort algorithms
→ …

tree



How to *store* a **tree**, in a DB table/relation??

"shredding"

table

node	fchild	nsib
1	2	-
2	-	3
3	-	4
4	5	-
5	-	6
6	-	-

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data

tree structured

→ how do we **store trees**?
→ can we benefit from **index structures**?
→ how to implement **tree navigation**?

Additional challenges posed by W3C's XQuery proposal.
→ a notion of **order**
→ a complex **type / schema system**
→ possibility to construct new tree nodes on the fly.

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data

tree structured

→ how do we **store trees**?
→ can we benefit from **index structures**?
→ how to implement **tree navigation**?

Additional challenges posed by W3C's XQuery proposal.
→ a notion of **order**
→ a complex **type / schema system**
→ possibility to construct new tree nodes on the fly.

XML = Threat to Databases… ?!

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

Some of these challenges are

Existing (DB) technology can**not** be applied to XML data

tree structured

→ how do we **store trees**?
→ can we benefit from **index structures**?
→ how to implement **tree navigation**?

Additional challenges posed by W3C's XQuery proposal.
→ a notion of **order**
→ a complex **type / schema system**
→ possibility to construct new tree nodes on the fly.

XML = Threat to Databases!!

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

You will learn about

→ Tree structured data (XML)
→ XML parsers & efficient memory representation

→ Query languages for XML (XPath, XQuery, XSLT…)
→ Efficient evaluation using finite-state automata

→ Mapping XML to databases
→ Advanced topics (query optimizations,
access control,
update languages…)

XML and Databases

This course will

→ introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

You will learn about

→ Tree structured data (XML)
→ XML parsers & efficient memory representation

→ Query Languages for XML (XPath, XQuery, XSLT…)
→ Efficient evaluation using finite-state automata

→ Mapping XML to databases
→ Advanced Topics (query optimizations, access control, update languages…)

You will NOT learn about

→Hacking CGI scripts
→HTML
→JavaScript
…

You NEED to program in

→ Java or
→ C++

About XML

→ XML is the World Wide Web Consortium's (W3C, `http://www.w3.org/`)
Extensible Markup Language

→ We hope to convince you that XML is not yet another hyped TLA, but is useful technology.

→ You will become best friends with one of the *most important data structures in Computing Science*, the **tree**.
XML is all about tree-shaped data.

→ You will learn to apply a number of closely related **XML standards**:

> Representing data: **XML** itself, **DTD**, **XMLSchema**, **XML** dialects
> Interfaces to connect PLs to XML: **DOM**, **SAX**
> Languages to query/transform XML: **XPath**, **XQuery**, **XSLT**.

About XML

We will talk about *algorithms and programming techniques* to efficiently manipulate XML data:

→ **Regular expressions** can be used to **validate** XML data

→ **Finite state automata** lie at the heart of highly efficient **XPath implementations**

→ **Tree traversals** may be used to preprocess XML trees in order to support **XPath evaluation**, to **store XML trees** in databases, etc.

In the end, you should be able to digest the thick pile of related W3C X___ standards.

(like, XQuery, XPointer, XLink, XHTML, XInclude, XML Base, XML Schema, …

Course Organization

Lecture Tuesday, 15:00 – 18:00
ChemicalSc M17 (ex AppliedSc) (K-F10-M17)

Lecturer Sebastian Maneth
Consult Friday, 11:00-12:00 (E508, L5)
All email to cs4317@cse.unsw.edu.au

Tutorial
Tuesday, 12:00-14:00 @ Quadrangle G040 (K-E15-G040) -- *before* lecture
Wednesday, 12:00-14:00 @ Quadrangle G022 (K-E15-G022)
Wednesday, 14:00-16:00 @ Quadrangle G022 (K-E15-G022)
Thursday, 14:00-16:00 @ Quadrangle G040 (K-E15-G040)
Thursday, 16:00-18:00 @ Quadrangle G022 (K-E15-G022)

Tutors ?, Kim Nguyen

All email to cs4317@cse.unsw.edu.au

Course Organization

Lecture Tuesday, 15:00 – 18:00
ChemicalSc M17 (ex AppliedSc) (K-F10-M17)

Lecturer Sebastian Maneth
Consult Friday, 11:00-12:00 (E508, L5)
All email to cs4317@cse.unsw.edu.au

Book None!

Suggested reading material:

Course slides of Marc Scholl, Uni Konstanz
http://www.inf.uni-konstanz.de/dbis/teaching/ws0506/database-xml/XMLDB.pdf

Theory / PL oriented, book draft:
http://arbre.is.s.u-tokyo.ac.jp/~hahosoya/xmlbook/

Course Organization

Lecture Tuesday, 15:00 – 18:00
ChemicalSc M17 (ex AppliedSc) (K-F10-M17)

Lecturer Sebastian Maneth
Consult Friday, 11:00-12:00 (E508, L5)
All email to cs4317@cse.unsw.edu.au

Programming Assignments

5 assignments, due every other Monday. (1st is due 15th March
2nd is due 29th March …)

Per assignment: 10 points (total: 60 points) (+2 bonus points)

Final Exam:

Final exam: 40 points (must get 16/40 to pass, that is 40%)

Outline - Assignments

You can freely choose to program your assignments in

→ C / C++, or
→ Java

However, your code **must compile with gcc / g++, javac**, as installed on CSE linux systems!

Submit code (using `give`) by Monday 23:59 (every other week)

Assignment 4 (harder) gets four weeks / counts double (20 Points)
due date 17th May

Outline - Assignments

1. Read XML, using DOM parser. Create document statistics	13 days
2. SAX Parse into memory structure: Tree vs DAG	2 weeks
3. Map XML into RDBMS	2 weeks (+1 week break)
4. XPath evaluation over main memory structures (+ *streaming* support)	4 weeks
5. XPath into SQL Translation	2 weeks

Outline - Lectures

1. Introduction to XML, Encodings, Parsers
2. Memory Representations for XML: Space vs Access Speed
3. RDBMS Representation of XML
4. DTDs, Schemas, Regular Expressions, Ambiguity
5. Node Selecting Queries: XPath
6. Efficient XPath Evaluation
7. XPath Properties: backward axes, containment test
8. Streaming Evaluation: how much memory do you need?
9. XPath Evaluation using RDBMS
10. Properties of XPath
11. XSLT
12. XQuery
13. Wrap up, Exam Preparation, Open questions, etc

Lecture 1

XML Introduction

Outline

1. Three **motivations for XML**
   (1) religious
   (2) practical
   (3) theoretical / mathematical

2. **Well-formed XML**

3. **Character Encodings**

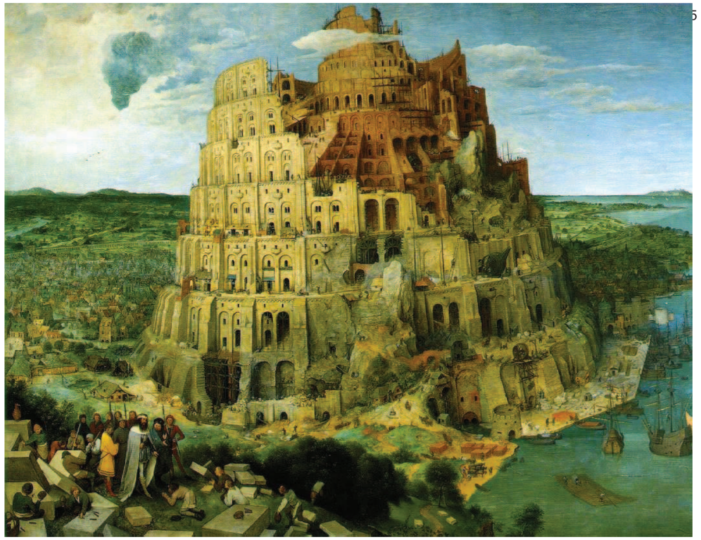
4. **Parsers for XML**

→ parsing into DOM (Document Object Model)

XML Introduction

Religious motivation for XML:

to have one language to speak about data.



XML Motivation (historical)

→ XML is a **Data Exchange Format**

1974 SGML (Charles Goldfarb at IBM Research)

1989 HTML (Tim Berners-Lee at CERN/Geneva)

1994 Berners-Lee founds Web Consortium (W3C)

1996 **XML** (W3C draft, v1.0 in 1998)

XML Motivation (historical)

→ XML is a **Data Exchange Format**

1974 SGML (Charles Goldfarb at IBM Research)

1989 HTML (Tim Berners-Lee at CERN/Geneva)

1994 Berners-Lee founds Web Consortium (W3C)

1996 **XML** (W3C draft, v1.0 in 1998)

`http://www.w3.org/TR/REC-xml/`

(2) Practical **XML** = data

```
Tom Henzinger
EPFL
Tom.Henzinger@epfl.ch

…

Helmut Seidl
TU Munich
seidl@inf.tum.de
```

Text file

(2) Practical **XML** = data + **structure**

```
Tom Henzinger
EPFL
Tom.Henzinger@epfl.ch

…

Helmut Seidl
TU Munich
seidl@inf.tum.de
```

Text file

"mark it up!"

```
<Related>
<colleague>
<name>Tom Henzinger</name>
<affil>EPFL</affil>
<email>
Tom.Henzinger@epfl.ch
</email>
</colleague>
…
<friend>
<name>Helmut Seidl</name>
<affil>TU Munich</affil>
<email>seidl@inf.tum.de
</email>
</friend>
</Related>
```

XML document

(2) Practical **XML** = data + **structure**

```
Tom Henzinger
EPFL
Tom.Henzinger@epfl.ch

…

Helmut Seidl
TU Munich
seidl@inf.tum.de
```

Text file

"mark it up!"

```
<Related>
<colleague>
<name>Tom Henzinger</name>
<affil>EPFL</affil>
<email>
Tom.Henzinger@epfl.ch
</email></colleague>
</colleague>
…
<friend>
<name>Helmut Seidl</name>
<affil>TU Munich</affil>
<email>seidl@inf.tum.de
</email>
</friend>
</Related>
```

XML document

Is this a good "template"?? What about last/first name?
Several affil's / email's…?

XML Documents

- → Ordinary text files (UTF-8, UTF-16, UCS-4 …)
- → Originates from typesetting/DocProcessing community
- → Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960’s)
- → Brackets describe a tree structure
- → Allows applications from different vendors to exchange data!

→ **standardized, extremely widely accepted!**

XML Documents

- → Ordinary text files (UTF-8, UTF-16, UCS-4 …)
- → Originates from typesetting/DocProcessing community
- → Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960’s)
- → Brackets describe a tree structure
- → Allows applications from different vendors to exchange data!

→ **standardized, extremely widely accepted!**

Social Implications!
All sciences (biology, geography, meteorology, astrology…) have own XML “dialects” to store *their* data optimally

XML Documents

- → Ordinary text files (UTF-8, UTF-16, UCS-4 …)
- → Originates from typesetting/DocProcessing community
- → Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960’s)
- → Brackets describe a tree structure
- → Allows applications from different vendors to exchange data!

→ **standardized, extremely widely accepted!**

Problem highly verbose, lots of repetitive markup, large files

XML Documents

- → Ordinary text files (UTF-8, UTF-16, UCS-4 …)
- → Originates from typesetting/DocProcessing community
- → Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960’s)
- → Brackets describe a tree structure
- → Allows applications from different vendors to exchange data!

→ **standardized, extremely widely accepted!**

Contra.. highly verbose, lots of repetitive markup, large files

Pro.. we have a standard! A Standard! A STANDARD!
→ ☺ *You never need to write a parser again! Use XML!* ☺

XML Documents

… instead of writing a parser, you simply fix your own “XML dialect”,

by describing all “admissible templates” (+ maybe even the specific

data types that may appear inside).

You do this, using an *XML Type definition language* such as DTD or Relax NG (Oasis).

Of course, such type definition languages are SIMPLE, because you want the parsers to be efficient!

They are similar to EBNF. → context-free grammar with reg. expr’s in the right-hand sides. ☺

XML Documents

Example **DTD** (Document Type Description)

```
Related     → (colleague | friend | family)*
colleague   → (name,affil*,email*)
friend      → (name,affil*,email*)
family      → (name,affil*,email*)
name        → (#PCDATA)
…
```

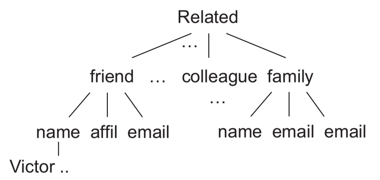
Element names and their content

XML Documents

Example DTD (Document Type Description)

```
Related     → (colleague | friend | family)*
colleague   → (name,affil*,email*)
friend      → (name,affil*,email*)
family      → (name,affil*,email*)
name        → (#PCDATA)
…
```

Element names and their content

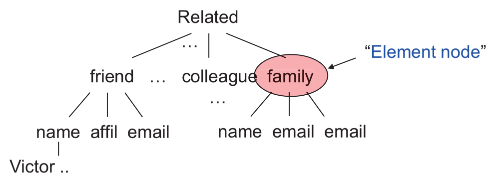


XML Documents

Example DTD

```
Related     → (colleague | friend | family)*
colleague   → (name,affil*,email*)
friend      → (name,affil*,email*)
family      → (name,affil*,email*)
name        → (#PCDATA)
…
```

Element names and their content

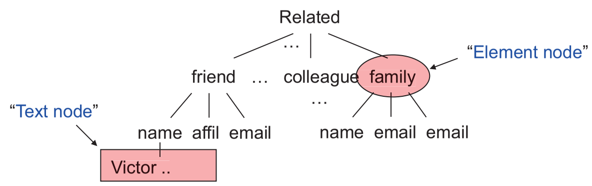


XML Documents

Example DTD

```
Related     → (colleague | friend | family)*
colleague   → (name,affil*,email*)
friend      → (name,affil*,email*)
family      → (name,affil*,email*)
name        → (#PCDATA)
…
```

Element names and their content



XML Documents

Example DTD

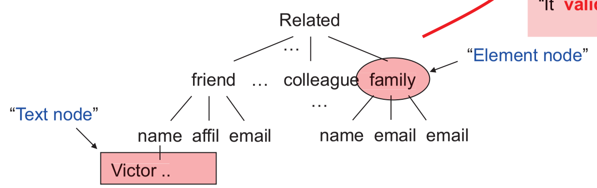
```
Related     → (colleague | friend | family)*
colleague   → (name,affil*,email*)
friend      → (name,affil*,email*)
family      → (name,affil*,email*)
name        → (#PCDATA)
…
```

Terminology

document is **valid** wrt the DTD

"It **validates**"

Element names and their content



XML Documents

What else: (besides *element* and *text* nodes)

→ attributes
→ processing instructions
→ comments
→ namespaces
→ entity references (two kinds)

XML Documents

What else: (besides *element* and *text* nodes)

→ attributes
→ processing instructions
→ comments
→ namespaces
→ entity references (two kinds)

<family rel="brother",age="25">
<name>
…
</family>

XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments
→ namespaces
→ entity references (two kinds)

<family rel=“brother”,age=“25”>
<name>
…
</family>

XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity references (two kinds)

<family rel=“brother”,age=“25”>
<name>
…
</family>

XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity references (two kinds)

<family rel=“brother”,age=“25”>
<name>
…
</family>

```
<!-- the 'price' element's namespace is http://ecommerce.org/schema -->
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```


XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity references (two kinds)

<family rel=“brother”,age=“25”>
<name>
…
</family>

```
<!-- the 'price' element's namespace is http://ecommerce.org/schema -->
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```


XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity *references* (two kinds)

character reference
Type <key>less-than</key> (<) to save options.

<family rel=“brother”,age=“25”>
<name>
…
</family>

```
<!-- the 'price' element's namespace is http://ecommerce.org/schema -->
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```


XML Documents

<?php sql (“SELECT * FROM …”) …?>

See 2.6 Processing Instructions

What else:

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity *references* (two kinds)

character reference
Type <key>less-than</key> (<) to save options.

<family rel=“brother”,age=“25”>
<name>
…
</family>

This document was prepared on &docdate; and

```
<!-- the 'price' element's namespace is http://ecommerce.org/schema -->
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```

Early Markup

The term markup has been coined by the **typesetting** community, *not* by computer scientist.

With the advent of printing press, writers and editors used (often marginal) notes to instruct printers to
→ Select certain fonts
→ Let passages of text stand out
→ Indent a line of text, etc

Proofreaders use a special set of symbols, their special **markup language**, to identify typos, formatting glitches, and similar erroneous fragments of text.

The markup language is designed to be easily recognizable in the actual flow of text.

Fortran 77 source, fixed form, space characters made explicit (␣)

Fortran 77

```
C␣THIS␣PROGRAM␣CALCULATES␣THE␣CIRCUMFERENCE␣AND␣AREA␣OF␣A␣CIRCLE␣WITH
C␣RADIUS␣R.
C
C␣DEFINE␣VARIABLE␣NAMES:
C␣␣␣␣␣␣R:␣RADIUS␣OF␣CIRCLE
C␣␣␣␣␣␣PI:␣VALUE␣OF␣PI=3.14159
C␣␣␣␣␣␣CIRCUM:␣CIRCUMFERENCE␣=␣2*PI*R
C␣␣␣␣␣␣AREA:␣AREA␣OF␣THE␣CIRCLE␣=␣PI*R*R
************************************************************************
C
␣␣␣␣␣␣REAL␣R,CIRCUM,AREA
C
␣␣␣␣␣␣PI␣=␣3.14159
C
C␣SET␣VALUE␣OF␣R:
␣␣␣␣␣␣R␣=␣4.0
C
C␣CALCULATIONS:
␣␣␣␣␣␣CIRCUM␣=␣2.*PI*R
␣␣␣␣␣␣AREA␣=␣PI*R*R
C
C␣WRITE␣RESULTS:
␣␣␣␣␣␣WRITE(6,*)␣␣'␣␣FOR␣A␣CIRCLE␣OF␣RADIUS',␣R,
␣␣␣␣␣+␣␣␣␣␣␣␣␣␣␣␣␣␣'␣␣THE␣CIRCUMFERENCE␣IS',␣CIRCUM,
␣␣␣␣␣+␣␣␣␣␣␣␣␣␣␣␣␣␣'␣␣AND␣THE␣AREA␣IS␣',␣AREA
C
␣␣␣␣␣␣END
```


An Application of Markup: A Comic Strip Finder

Problem:

- **Query a database of comic strips by content**. We want to approach the system with queries like:
  1. *Find all strips featuring Dilbert but not Dogbert.*
  2. *Find all strips with Wally being angry with Dilbert.*
  3. *Show me all strips featuring characters talking about XML.*

Approach:

- Unless we have next^n generation image recognition software available, we obviously have to **annotate** the comic strips to be able to process the queries above:

strips	bitmap	annotation
	⋮	⋮
		...Dilbert...Dogbert Wally...
	⋮	⋮

Early Markup

Computer scientists adopted the markup idea – originally to **annotate program source code**:

→ Design the markup language such that its constructs are **easily recognizable** by a machine.

→ **Approaches**
(1) Markup is written using a **special set of characters**, disjoint from the set of characters that form the tokens of the program
(2) Markup occurs in places in the source file where program code may not appear (**program layout**).

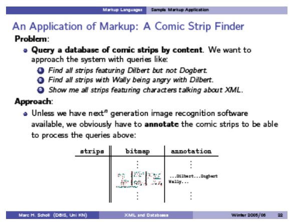
Example: Fortran 77 fixed form source:

→ Fortran statements start in column 7 and do not exceed column 72,
→ a Fortran statement longer than 66 chars may be continued on the next line
If a character not in { 0,!,_ } is place in column 6 of the continuing line
→ comment lines start with a "C" or "*" in column 1,
→ Numeric labels (DO, FORMAT statements) have to be placed in columns 1-5.

Sample Markup Application

A Comic Strip Finder

→ Next 8 slides from Marc Scholl's 2005 lecture.


Markup Languages | Sample Markup Application
An Application of Markup: A Comic Strip Finder
Problem:
Query a database of comic strips by content. We want to approach the system with queries like:
Find all strips featuring Dilbert but not Dogbert.
Find all strips with Wally being angry with Dilbert.
Show me all strips featuring characters talking about XML.
Approach:
Unless we have next^n generation image recognition software available, we obviously have to annotate the comic strips to be able to process the queries above:
strips | bitmap | annotation
...Dilbert...Dogbert Wally...
Marc H. Scholl (DBIS, Uni KN) | XML and Databases | Winter 2005/06 | 22

Stage 1: ASCII-Level Markup

ASCII-Level Markup

```
Pointy-Haired Boss: >>Speed is the key to success.<<
Dilbert: >>Is it okay to do things wrong if we're really, really fast?<<
Pointy-Haired Boss: >>Um... No.<<
Wally: >>Now I'm all confused.  Thank you very much.<<
```

- ASCII C0 character sequence 0x0d, 0x0a (CR, LF) divides lines,
- each line contains a character name, then a colon (:), then a line of speech (comic-speak: *bubble*),
- the contents of each bubble are delimited by >> and <<.

✎ Which kind of queries may we ask now?
And what kind of software do we need to complete the comic strip finder?

Stage 2: HTML-Style Physical Markup

dilbert.html

```
<h1>Dilbert</h1>
<h2>Panel 1</h2>
<ul>
   <li> <b>Pointy-Haired Boss</b> <em>Speed is the key
        to success.</em>
</ul>
<h2>Panel 2</h2>
<ul>
   <li> <b>Dilbert</b> <em>Is it okay to do things wrong
        if we're really really fast?</em>
</ul>
<h2>Panel 3</h2>
<ul>
   <li> <b>Pointy-Haired Boss</b> <em>Um... No.</em>
   <li> <b>Wally</b> <em>Now I'm all confused.
        Thank you very much.</em>
</ul>
```


HTML: Observations

- HTML defines a number of markup **tags**, some of which are required to match (<*t*>. . . </*t*>).
- Note that HTML tags primarily describe **physical markup** (font size, font weight, indentation, . . .)
- Physical markup is of limited use for the comic strip finder (the **tags do not reflect the structure of the comic content**).

Stage 3: XML-Style Logical Markup

- We create a **set of tags that is customized** to represent the content of comics, *e.g.*:

  `<character>Dilbert</character>`
  `<bubble>Speed is the key to success.</bubble>`

- New types of queries may require new tags: No problem for XML!
  - Resulting set of tags forms a new markup language (**XML dialect**).
- **All** tags need to appear in **properly nested** pairs (*e.g.*, <*t*>. . . <*s*>. . . </*s*>. . . </*t*>).
- Tags can be freely nested to reflect the **logical structure** of the comic content.

✎ Parsing XML?

In comparison to the stage 1 ASCII-level markup parsing, how difficult do you rate the construction of an XML parser?

In our example

dilbert.xml

```
<strip>
  <panel>
    <speech>
      <character>Pointy-Haired Boss</character>
      <bubble>Speed is the key to success.</bubble>
    </speech>
  </panel>
  <panel>
    <speech>
      <character>Dilbert</character>
      <bubble>Is it okay to do things wrong
              if we're really, really fast?</bubble>
    </speech>
  </panel>
  <panel>
    <speech>
      <character>Pointy-Haired Boss</character>
      <bubble>Um... No.</bubble>
    </speech>
    <speech>
      <character>Wally</character>
      <bubble>Now I'm all confused.
              Thank you very much.</bubble>
    </speech>
  </panel>
</strip>
```


Stage 4: Full-Featured XML Markup

- Although fairly simplistic, the previous stage clearly constitutes an improvement.
- XML comes with a number of additional constructs which allow us to convey even more useful information, *e.g.*:
  - **Attributes** may be used to qualify tags (avoid the so-called *tag soup*).
    Instead of
    - `<question>Is it okay ...?</question>`
      `<angry>Now I'm ...</angry>`

    use
    - `<bubble tone="question">Is it okay ...?</bubble>`
      `<bubble tone="angry">Now I'm ...</bubble>`
  - **References** establish links internal to an XML document:
    Establish link target:
    - `<character id="phb">The Pointy-Haired Boss</character>`

    Reference the target:
    - `<bubble speaker="phb">Speed is the key to success.</bubble>`

dilbert.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>
<strip copyright="United Feature Syndicate" year="2000">
  <prolog>
    <series href="http://www.dilbert.com/">Dilbert</series>
    <author>Scott Adams</author>
    <characters>
      <character id="phb">The Pointy-Haired Boss</character>
      <character id="dilbert">Dilbert, The Engineer</character>
      <character id="wally">Wally</character>
      <character id="alice">Alice, The Technical Writer</character>
    </characters>
  </prolog>
  <panels length="3">
    <panel no="1">
      <scene visible="phb">
        Pointy-Haired Boss pointing to presentation slide.
      </scene>
      <bubbles>
        <bubble speaker="phb">Speed is the key to success.</bubble>
      </bubbles>
    </panel>
    <panel no="2">
      <scene visible="wally dilbert alice">
        Wally, Dilbert, and Alice sitting at conference table.
      </scene>
      <bubbles>
        <bubble speaker="dilbert" to="phb" tone="question">
          Is it ok to do things wrong if we're really, really fast?
        </bubble>
      </bubbles>
    </panel>
    <panel no="3">
      <scene visible="wally dilbert">Wally turning to Dilbert, angrily.
      </scene>
      <bubbles>
        <bubble speaker="phb" to="dilbert">Um... No.</bubble>
```

61

Today, XML has many friends:

Query Languages

XPath, XSLT, XQuery, …   (mostly by W3C)

Implementations   (Parsers, Validators, Translators)

SAX, Xalan, Galax, Xerxes, …

(by IBM/Apache, Microsoft, Oracle, Sun…)

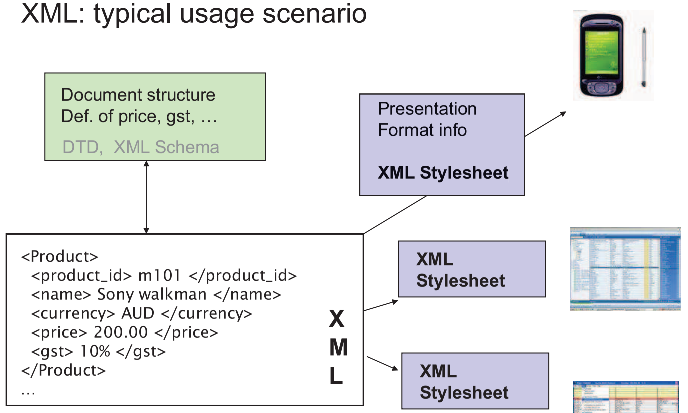
Current Issues

- DB/PL support   (“data binding”, JBind, Castor, Zeus…)

- storage support   (compression, data optimization)

62

XML: typical usage scenario



One data source   → several (dynamic gen.) **views**

63

XML: where is it used ?

- Document formats:

SVG (vector images), OpenDocument Format (OpenOffice, GoogleDocs), DocBook (Text formatting), EPUB (electronic books), XHTML (web), …

- Protocol formats:

SOAP (WebServices), XMPP (Jabber, GoogleTalk), AJAX (custom XML messages used for interactive web sites: Facebook, Gmail, Tweeter,…), RSS feeds (used for blogs/news sites), …

- Custom data format:

Bio-informatics, Linguistics, Configuration files, Geographic data (OpenStreetMap, Google maps), Bibliographic Resources (MedLine, ADC)

64

(3) Theoretical / Mathematical

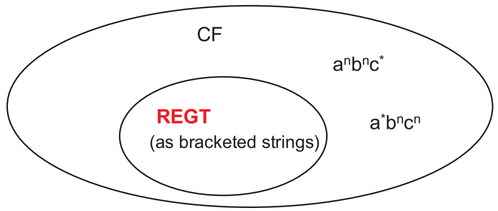
Regular Tree languages   (REGT).

Many characterizations: *Reg. Tree Grammars*
Tree Automata
MSO Logic

Context-Free:

```
E ::= E '+' E
E ::= E 'x' E
E ::= [1-9][0-9]*
```

Nice properties: *Closed under intersection   (union, complement)*
Decidable equivalence



65

2. Well-Formed XML

From the W3C XML recommendation

`http://www.w3.org/TR/REC-xml/`

“A textual object is **well-formed XML** if,

(1) taken as a whole, it **matches the production labeled** *document*
(2) it meets all the **well-formedness constraints** given in this specification ..”

---

document = start symbol of a context-free grammar   (“XML grammar”)

→ (1) contains the *contex-free properties* of well-formed XML
→ (2) contains the *context-dependent properties* of well-formed XML

There are 10 WFCs (well-formedness constraints).
E.g.: **Element Type Match**   “The Name in an element’s end tag must match the element name in the start tag.”

66

2. Well-Formed XML

From the W3C XML recommendation

`http://www.w3.org/TR/REC-xml/`

“A textual object is **well-formed XML** if,

(1) taken as a whole, it **matches the production labeled** *document*
(2) it meets all the **well-formedness constraints** given in this specification ..”

---

document = start symbol of a context-free grammar   (“XML grammar”)

→ (1) contains the *contex-free properties* of well-formed XML
→ (2) contains the *context-dependent properties* of well-formed XML

There are 10 WFCs (well-formedness constraints).
E.g.: **Element Type Match**   “The Name in an element’s end tag must match the element name in the start tag.”
→ Why is this *not* context-free?

2. Well-Formed XML

Context-free grammar in EBNF = System of production rules of the form

`lhs ::= rhs`

`lhs` a nonterminal symbol (e.g, *document*)
`rhs` a string over nonterminal and terminal symbols.

Additionally (EBNF), we may use regular expressions in `rhs`.
Such as:

`r*`	denoting	`ε, r, rr, rrr, …`	zero or more repititions
`r+`	denoting	`rr*`	one or more repititions
`r?`	denoting	`r | ε`	optional r
`[abc]`	denoting	`a | b | c`	character class
`[a-z]`	denoting	`a | b | … | z`	character class

XML Grammar - EBNF-style

```
[1]  document     ::= prolog element Misc*
[2]      Char     ::= a Unicode character
[3]         S     ::= (' ' | '\t' | '\n' | '\r')+
[4]  NameChar     ::= (Letter | Digit | '.' | '-' | ':'
[5]      Name     ::= (Letter | '_' | ':') (NameChar)*

[22]    prolog    ::= XMLDecl? Misc* (doctypedecl Misc*)?
[23]   XMLDecl    ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]VersionInfo   ::= S'version'Eq("'"VersionNum"'"|'"'VersionNum'"')
[25]        Eq    ::= S? '=' S?
[26]VersionNum    ::= '1.0'

[39]   element    ::= EmptyElemTag
                    | STag content Etag
[40]      STag    ::= '<' Name (S Attribute)* S? '>'
[41] Attribute    ::= Name Eq AttValue
[42]      ETag    ::= '</' Name S? '>'
[43]   content    ::= (element | Reference | CharData?)*
[44]EmptyElemTag  ::= '<' Name (S Attribute)* S? '/>'

[67] Reference    ::= EntityRef | CharRef
[68] EntityRef    ::= '&' Name ';'
[84] Letter       ::= [a-zA-Z]
[88] Digit        ::= [0-9]
```


XML Grammar - EBNF-style

As usual, the XML grammar can be systematically transformed into a program, an **XML parser**, to be used to check the syntax of XML input

Parsing XML

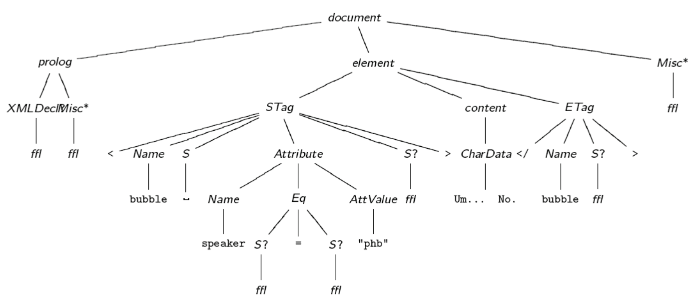
1. Starting with the symbol **`document`**, the parser uses the `lhs::=rhs` rules to expand symbols, constructing a **parse tree**.

2. Leaves of the parse tree are characters which have no further expansion

3. The XML input is **parsed** successfully if it perfectly matches the parse tree's **front** (concatenate the parse tree's leaves from left-to-right, while removing ε symbols).

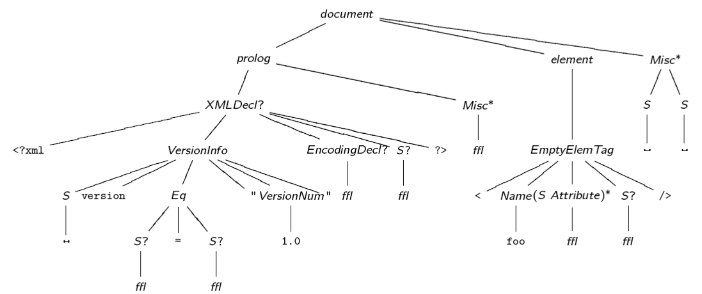
Example 1
Parse tree for XML input

`<bubble speaker="phb">Um... No.</bubble>`



Example 2
Parse tree for XML input

`<?xml version="1.0"?><foo/>`



2. Well-Formed XML

Terminology

- **tag names** `name`, `email`, `author`, …
- **start tag** `<name>`, **end tag** `</name>`

- **elements** `<name>` … `</name>`, `<author age="99">` … `</author>`
- elements may be **nested**
- **empty element** `<red></red>` is abbreviated as `<red/>`

- an XML document: **single root element**

  `<someTag>` …. `</someTag>`

- **well-formed constraints**
  → begin/end tags match
  → no attribute name may appear more than once in a start tag or empty element tag
  → a parsed entity must not contain a recursive reference to itself, either directly or indirectly

context-dependent properties

Well-formed XML (fragments)

```
<Staff>
  <Name>
    <FirstName> Sebastian </FirstName>
    <LastName> Maneth </LastName>
  </Name>
  <Login> smaneth </Login>
  <Ext> 2481 </Ext>
</Staff>
```

a `Name` element

a `Staff` element

Non-well-formed XML

```
<foo> oops </bar>
<foo> oops </Foo>
<foo> oops .. <EOT>
<a><b><c></a></b></c>
```


Well-formed XML (fragments)

```
<Staff>
  <Name>
    <FirstName> Sebastian </FirstName>
    <LastName> Maneth </LastName>
  </Name>
  <Login> smaneth </Login>
  <Ext> 2481 </Ext>
</Staff>
```

a `Name` element

a `Staff` element

Non-well-formed XML

```
<foo> oops </bar>
<foo> oops </Foo>
<foo> oops .. <EOT>
<a><b><c></a></b></c>
```

Questions

How can you implement the three well-formed constraints?

When, during parsing, do you apply the checks?

Character Encoding

- For a computer, a character like `X` is nothing but an 8 (16/32) bit **number** whose value is *interpreted* as the character `X`, when needed.
- Problem: many such number → character mappings, the so called **encodings** are in use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese …), **conflicting intersections** between encodings are common.

Example

`0xcb 0xe4 0xd3` —iso-8859-7→ Æ δ Σ

`0xcb 0xe4 0xd3` —iso-8859-15→ Ë ä Ó

Unicode

- The Unicode `http://www.unicode.org` Initiative aims to define a new encoding that tries to embrace all character needs.
- The Unicode encoding contains characters of "all" languages of the world plus scientific, mathematical, technical, box drawing, … symbols
- Range of the Unicode encoding: `0x0000-0x10FFFF` `(=16*65536)`

→ Codes that fit into the first 16 bits (denoted `U+0000-U+FFFF`) encode the most widely used languages and their characters (**Basic Multilingual Plane, BMP**)

→ Codes U+0000-U+007F have been assigned to match the 7-bit ASCII encoding which is pervasive today.

Unicode Transformation Formats

Current CPUs operate most efficiently on **32-bit words** (16-bit words, bytes)

Unicode thus developed Unicode Transformation Formats (UTFs) which define how a Unicode character code between `U+0000` and `U+10FFFF` is to be mapped into a 32-bit word (16-bit word, byte).

UTF-32
→ Simply map exactly to the corresponding 32-bit value
→ For each Unicode character in UTF-32: waste of at least 11 bits!

Unicode Transformation Formats

Current CPUs operate most efficiently on **32-bit words** (16-bit words, bytes)

Unicode thus developed Unicode Transformation Formats (UTFs) which define how a Unicode character code between `U+0000` and `U+10FFFF` is to be mapped into a 32-bit word (16-bit word, byte).

UTF-32
→ Simply map exactly to the corresponding 32-bit value
→ For each Unicode character in UTF-32: waste of at least 11 bits!

UTF-16
Map a Unicode character into **one or two 16-bit words**
→ U+0000 to U+FFFF map exactly to the corresponding 16-bit value
→ above U+FFFF: subtract 0x010000 and then fill the □'s in
1101 10□□ □□□□ □□□□ 1101 11□□ □□□□ □□□□

E.g. Unicode character `U+012345` `(0x012345 - 0x010000 = 0x02345)`

UTF-16: 1101 10<u>00 0000 1000</u> 1101 11<u>11 0100 0101</u>

Unicode Transformation Formats

•**UTF-16** works correctly, because the character codes between

1101 10□□ □□□□ □□□□ and

1101 11□□ □□□□ □□□□ (with each □ replaced by a 0)

are left unassigned in Unicode!!! (range 0xD800 – 0xDFFF is reserved)

Unicode Transformation Formats

•**UTF-16** works correctly, because the character codes between

1101 10□□ □□□□ □□□□ and

1101 11□□ □□□□ □□□□ (with each □ replaced by a 0)

are left unassigned in Unicode!!! (range 0xD800 – 0xDFFF is reserved)

•**UTF-16** is the **only** encoding in Java: a `char` is **16 bits** large, **not** 8 as in C/C++

WARNING: a "single" character which uses two 16 bits fields is made up of 2 chars!

Example, the string "𝄞" takes two 16 bit fields:

```
String s = "\uD834\uDD1E";
s.length();       // returns 2
s.charAt(1);      //returns \uDD1E, it's an INVALID character.
s.codePointAt(1); //returns the integer 0xD834DD1E
```


UTF-8

Maps a unicode character into **1, 2, 3, or 4 bytes**.

Unicode range	Byte sequence
U+000000 → U+00007F	0□□□□□□□
U+000080 → U+0007FF	110□□□□□ 10□□□□□□
U+000800 → U+00FFFF	1110□□□□ 10□□□□□□ 10□□□□□□
U+010000 → U+10FFFF	11110□□□ 10□□□□□□ 10□□□□□□ 10□□□□□□

Spare bits (□) are filled from right to left. Pad to the left with 0-bits.

E.g. U+00A9 in UTF-8 is 11000010 10101001
U+2260 in UTF-8 is 11100010 10001001 10100000

UTF-8

→ For a UTF-8 multi-byte sequence, the length of the sequence is equal to the number of leading 1-bits (in the first byte)

→ Character boundaries are simple to detect

→ UTF-8 encoding does not affect (binary) sort order

→ Text processing software designed to deal with 7-bit ASCII remains functional.

(especially true for the C programming language and its string (`char[]`) representation)

XML and Unicode

→ A conforming XML parser is **required** to correctly process UTF-8 and UTF-16 encoded documents. (The W3C XML Recommendation predates the UTF-32 definition)

→ Documents that use a different encoding must announce so using the XML text declaration, e.g.,

`<?xml encoding="iso-8859-15"?>`
or `<?xml encoding="utf-32"?>`

→ Otherwise, an XML parser is encouraged to **guess** the encoding while reading the very first bytes of the input XML document:

Head of doc (bytes)	Encoding guess
0x00 0x3C 0x00 0x3F	UTF-16 (*little Endian*)
0x3C 0x00 0x3F 0x00	UTF-16 (*big Endian*)
0x3c 0x3F 0x78 0x6D	UTF-8 (or ASCII)

Notice: < = U+003C, ? = U+003F, x = U+0078, m = U+006D

XML and Unicode

Questions

→ What does "*guess the encoding*" mean? Under which circumstances does the parser **know** it has determined the correct encoding?

Are there cases when it canNOT determine the correct encoding?

→ What about efficiency of the UTFs? For different texts, compare the space requirement in UTF-8/16 and UTF-32 against each other. Which characters do you find above 0xFFFF in Unicode?

Can you imagine a scenario where UTF-32 is *faster* than UTF-8/16?

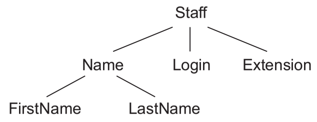
85

The XML Processing Model

→ On the **physical** side, XML defines nothing but a **flat text format**, i.e., it defines a set of (e.g. UTF-8/16) character sequences being *well-formed XML*.

→ Applications that want to analyze and transform XML data in any meaningful way will find processing flat character sequences hard and inefficient!

→ The nesting of XML elements and attributes, however, defines a **logical tree-like structure**.



86

The XML Processing Model

→ Virtually all **XML applications operate on the logical tree view** which is provided to them by an XML processor (i.e., "parse & store").

→ XML processors are widely available (e.g., Apache's Xerces).

How is the XML processor supposed to communicate the XML tree structure to the application?

87

The XML Processing Model

→ Virtually all **XML applications operate on the logical tree view** which is provided to them by an XML processor (i.e., "parse & store").

→ XML processors are widely available (e.g., Apache's Xerces).

How is the XML processor supposed to communicate the XML tree structure to the application?

→ For many PL's there are "data binding" tools. Gives very flexible way to get PL view of the XML tree structure.

88

The XML Processing Model

→ Virtually all **XML applications operate on the logical tree view** which is provided to them by an XML processor (i.e., "parse & store").

→ XML processors are widely available (e.g., Apache's Xerces).

How is the XML processor supposed to communicate the XML tree structure to the application?

→ For many PL's there are "data binding" tools. Gives very flexible way to get PL view of the XML tree structure.

But first, let's see what the standard says…

89

The XML Processing Model

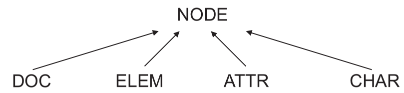
→ Virtually all **XML applications operate on the logical tree view** which is provided to them by an XML processor (i.e., "parse & store").

→ XML processors are widely available (e.g., Apache's Xerces).

How is the XML processor supposed to communicate the XML tree structure to the application?

→ through a fixed interface of accessor functions: The XML Information Set

The accessor functions operate on different types of node objects:



90

Accessor Functions ("node properties")

Node type	Property	Comment

DOC	`childern: DOC→ELEM`
	`base-uri: DOC→STRING`
	`version : DOC→STRING`
		root element

ELEM	`localname : ELEM→STRING`
	`children : ELEM→[NODE]`
	`attributes: ELEM→[ATTR]`
	`parent : ELEM→NODE`
		[..]=sequence type

ATTR	`localname : ATTR→STRING`
	`value : ATTR→STRING`
	`owner : ATTR→ELEM`	

CHAR	`code : CHAR→UNICODE` `parent : CHAR→ELEM`	a single character

XML Information Set - `http://www.w3.org/TR/xml-infoset`

Information set of a sample document

```
<?xml version="1.0"?>
<forecast date="Thu, May 16">
  <condition>sunny</condition>
  <temperature unit="Celsius">23</temperature>
</forecast>
```

```
children(d) = e1
base-uri(d) = "file:/…"
version(d)  = "1.0"

localname(e1) = "forecast"
children(e1)  = [e2,e3]
attributes(e1)= [a1]
parent(e1)    = d

localname(a1) = "date"
value(a1)     = "Thu, May 16"
owner(a1)     = e1

localname(e2) = "condition"
children(e2)  = [c1,c2,c3,c4,c5]
attributes(e2)= []
parent(e2)    = e1
```

```
code(c1)   = U+0073   (='s')
parent(c1)= e2
  . . .
code(c5)   = U+0079   (='y')
parent(c5)= e2

localname(e3) ="temperature"
children(e3)  =[c6,c7]
attributes(e3)=[a2]
parent(e3)    =a1
  . . .
```


Questions

(1) A NODE type can be one of DOC, ELEM, ATTR, or CHAR.
In the two places of the property functions where NODE appears, which of the four types may *actually* appear there?

For instance, is this allowed?

```
localname(e1) = "condition"
children(e1)  = [c1,e2,c2]
```

(2) What about WHITESPACE?
Where in an XML document does it matter, and where not?

Where in the Infoset Example (previous slide)
are the returns and indentations of the document?
(did we do a mistake? If so, what is the correct Infoset?)

Querying the Infoset

Using the Infoset, we can analyze a given XML document in many ways.
For instance:

→ Find all ELEM nodes with localname=`bubble`, owning an ATTR node with localname=`speaker` and value=`Dilbert`.

→ List all `panel` ELEM nodes containing a bubble spoken by "`Dogbert`"

→ Starting in panel 2 (ATTR `no`), find all bubbles following those spoken by "Alice"

Such queries appear very often and can conveniently be described using XPath queries:

→ `//bubble/[@speaker="Dilbert"]`
→ `//panel[.//bubble[@speaker="Dogbert"]]`
→ `//panel[@no="2"]//bubble[@speaker="Alice"]/following::bubble`

3. Parsers for XML

Two different approaches:

(1) Parser stores document into a fixed (standard) data structure (e.g., an Infoset compliant data structure, such as DOM)

```
parser.parse("foo.xml");
doc = parser.getDocument ();
```

(2) Parser triggers "events". Does not store!
User has to write own code on how to store / process the events triggered by the parser.

DOM = Document Object Model

→ W3C standard,
see `http://www.w3.org/TR/REC-DOM-Level-1/`

DOM – Document Object Model

→ Language and platform-independent view of XML

→ DOM APIs exist for many PLs (Java, C++, C, Perl, Python, …)

DOM relies on two main concepts

(1) The XML processor constructs the **complete XML document tree** (in-memory)

(2) The XML application issues DOM library calls to **explore** and **manipulate** the XML tree, or to **generate** new XML trees.

Advantages
- easy to use
- once in memory, no tricky issues with XML syntax anymore
- all DOM trees serialize to well-formed XML (even after arbitrary updates)!

DOM – Document Object Model

→ Language and platform-independent view of XML

→ DOM APIs exist for many PLs (Java, C++, C, Perl, Python, …)

DOM relies on two main concepts

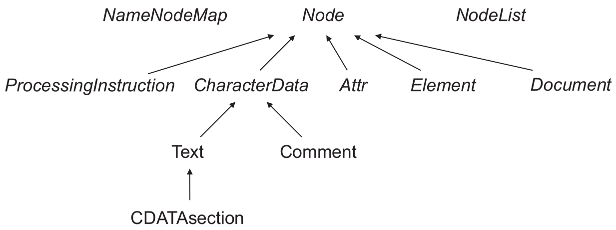
(1) The XML processor constructs the **complete XML document tree** (in-memory)

(2) The XML application issues DOM library calls to **explore** and **manipulate** the XML tree, or to **generate** new XML trees.

Advantages
- easy to use
- once in memory, no tricky issues with XML syntax arise anymore
- all DOM trees serialize to well-fromed XML (even after arbitrary updates)!

Disadvantage Uses LOTS of memory!!

DOM Level 1 (Core)



Character strings (DOM type *DOMString*) are defined to be encoded using UTF-16 (e.g., Java DOM reresents type *DOMString* using its `String` type).

DOM Level 1 (Core)

Some methods

```
DOM type   Method                         Comment

Node       nodeName   } : DOMString       redefined in subclasses
           nodeValue  }
           parentNode   : Node
           firstChild   : Node            leftmost child
           nextSibling  : Node            returns NULL for root elem
                                          or last child or attributes
           childNodes   : NodeList
           attributes   : NamedNodeMap
           ownerDocument: Document
           replaceChild : Node

Document   createElement : Element        creates element with
                                            given tag name
           createComment : Comment
           getElementsByTagName: NodeList   list of all Elem nodes
                                            in document order
```


DOM Level 1 (Core)

Name,Value, and attributes depend on the **type** of the current node.

The values of `nodeName`, `nodeValue`, and `attributes` vary according to the node type as follows:

	nodeName	nodeValue	attributes
Element	tagName	null	NamedNodeMap
Attr	name of attribute	value of attribute	null
Text	#text	content of the text node	null
CDATASection	#cdata-section	content of the CDATA Section	null
EntityReference	name of entity referenced	null	null
Entity	entity name	null	null
ProcessingInstruction	target	entire content excluding the target	null
Comment	#comment	content of the comment	null
Document	#document	null	null
DocumentType	document type name	null	null
DocumentFragment	#document-fragment	null	null
Notation	notation name	null	null

DOM Level 1 (Core)

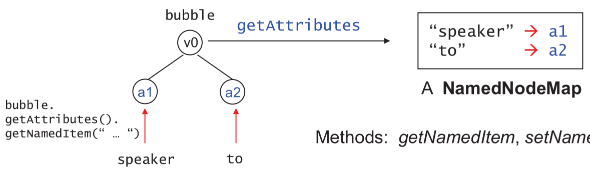
Some details

Creating an *element/attribute* using *createElement/createAttribute* does not wire the new node with the XML tree structure yet.
→Call *insertBefore*, *replaceChild*, …, to wire a node at an explicity position

DOM type *NodeList* makes up fo the lack of collection data types in most programming languages

DOM type *NamedNodeMap* represents an *association table* (nodes may be accessed by name)

Example:



A **NamedNodeMap**

Methods: *getNamedItem*, *setNamedItem*,…

E.g. Find all occurrences of `Dogbert` speaking (attribute `speaker` of element `bubble`)

dogbert.cc (1)

```
// Xerces C++ DOM API support
#include <dom/DOM.hpp>
#include <parsers/DOMParser.hpp>

void dogbert (DOM_Document d)
{
  DOM_NodeList     bubbles;
  DOM_Node         bubble, speaker;
  DOM_NamedNodeMap attrs;

  bubbles = d.getElementsByTagName ("bubble");

  for (unsigned long i = 0; i < bubbles.getLength (); i++) {
    bubble = bubbles.item (i);

    attrs = bubble.getAttributes ();
    if (attrs != 0)
      if ((speaker = attrs.getNamedItem ("speaker")) != 0)
        if (speaker.getNodeValue ().
              compareString (DOMString ("Dogbert")) == 0)
          cout << "Found Dogbert speaking." << endl;
  }
}
```


Questions

Given an XML file of, say, 50K, how large is its DOM representation in main memory?

How much larger, in the *worst case*, is a DOM representation with respect to the size of the XML document? (difficult!)

How could we decrease the memory need of DOM, while preserving its functionality?

END
Lecture 1

Next

“Even trigger” Parsers for XML:

→ Build your own XML data structure
and fill it up as the parser triggers input “events”.