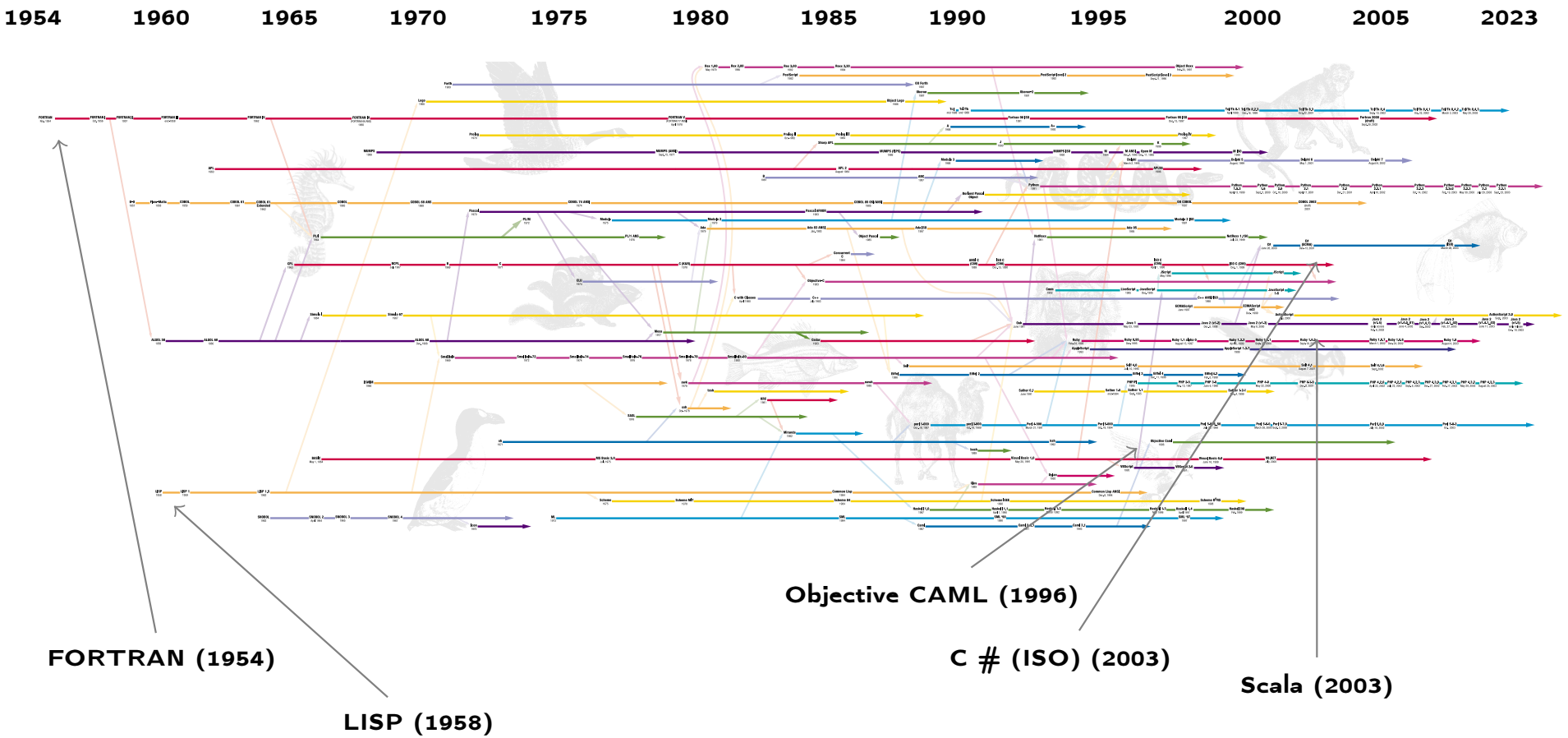
Course: XML Essentials

Pierre Genevès
CNRS

University Grenoble Alpes

Programming Languages



Even more “DSLs”, e.g. SPARQL (2008), Julia (2009), XQuery (2010)
Frequent renewal, rapid evolution

Data

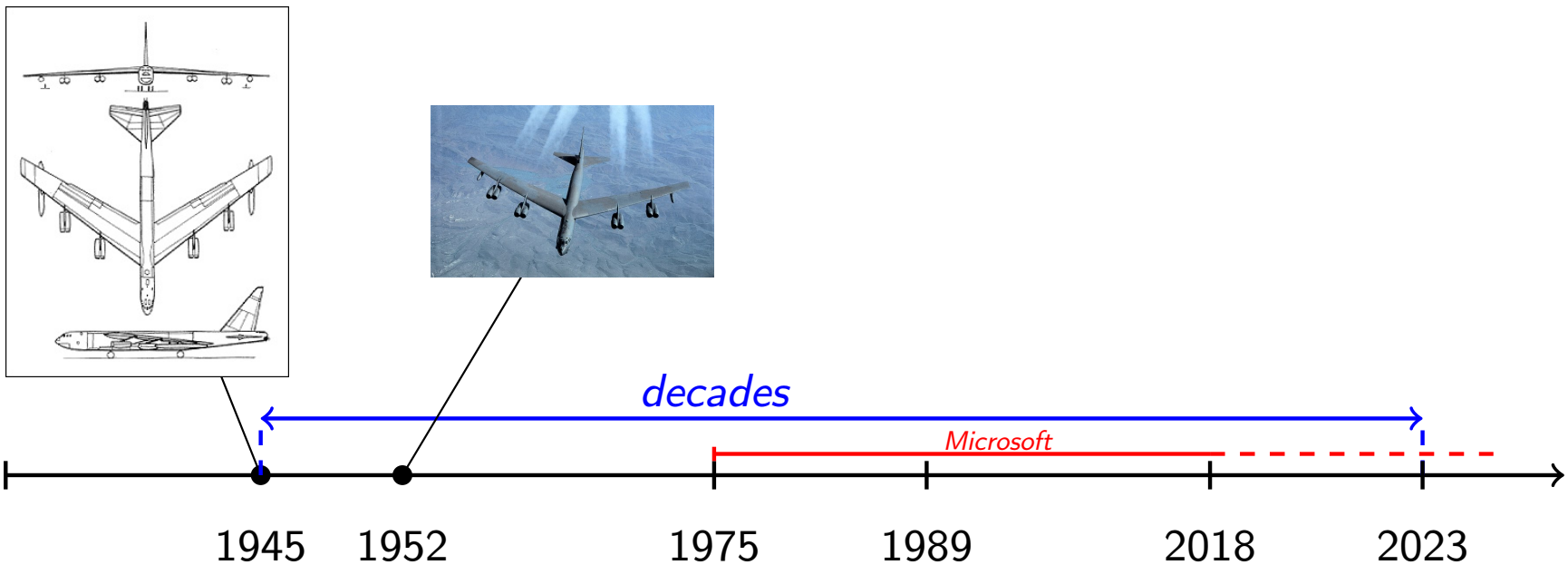
- Often, data are more important than programs

Data

- Often, data are more important than programs
- One reason for this is that data often have a much longer life cycle than programs
- Examples: banking, aeronautical technical documentation, etc.

Example: Aeronautical Technical Documentation

In aeronautics, it is common to find products with life cycles that last for several decades, *e.g.* the B-52:



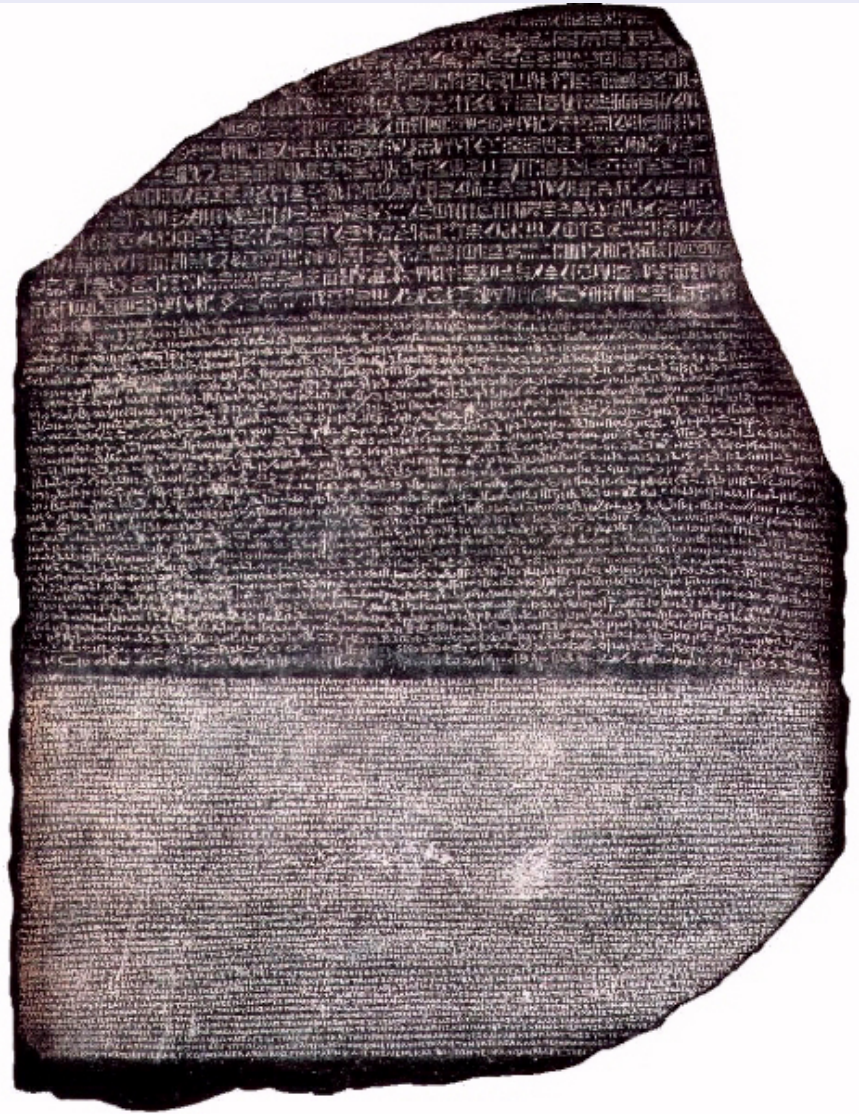
→ **How to ensure long-term access to data?**

→ How to design systems such that manipulated data can still be read 15 or 50 years later?

→ **How to ensure long-term access to data?**

→ How to design systems such that manipulated data can still be read 15 or 50 years later?

→ An old concern...



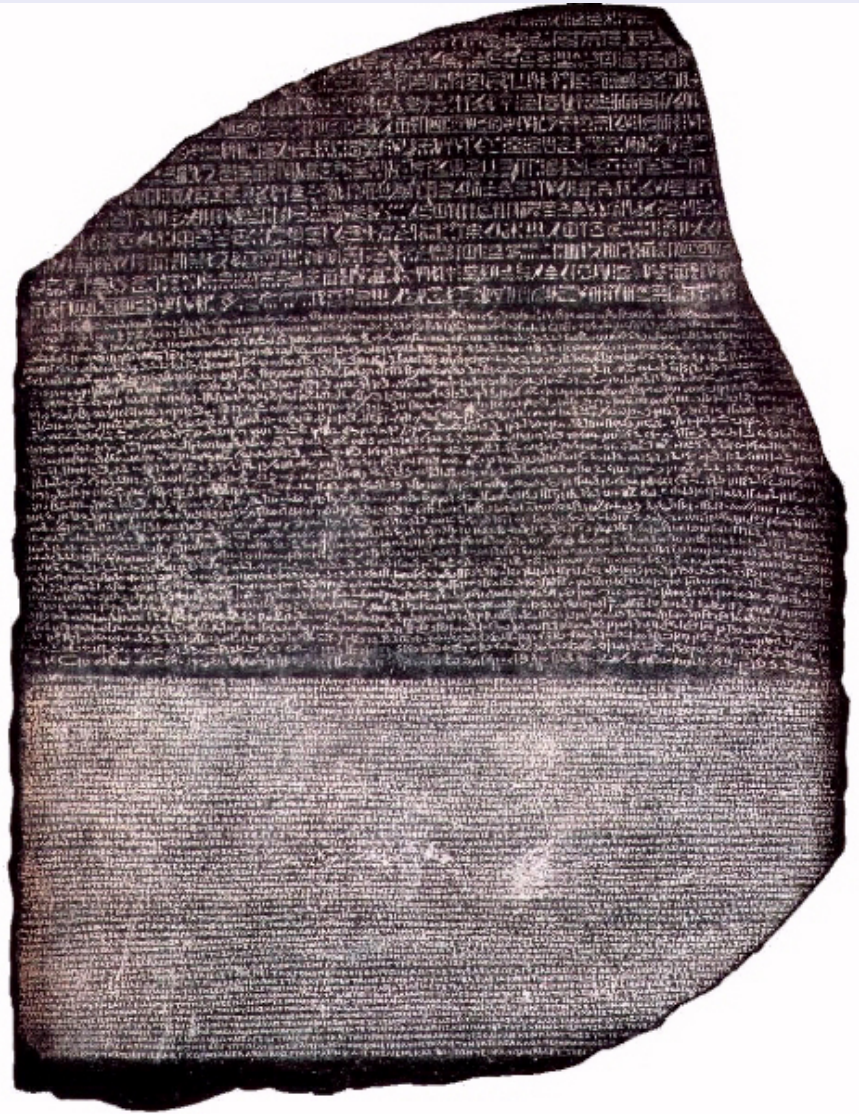
La pierre de Rosette (The Rosetta Stone).

→ **How to ensure long-term access to data?**

→ How to design systems such that manipulated data can still be read 15 or 50 years later?

→ An old concern...

- Can we really do better with computers?



La pierre de Rosette (The Rosetta Stone).

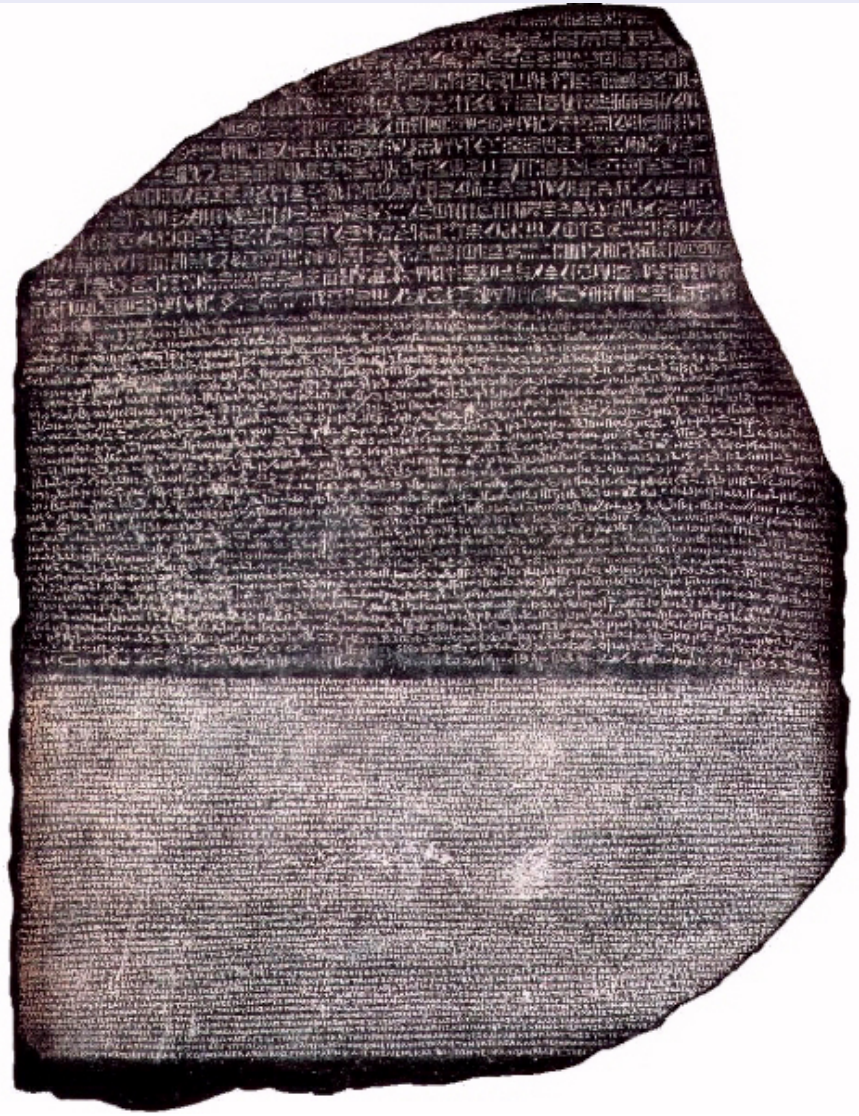
→ **How to ensure long-term access to data?**

→ How to design systems such that manipulated data can still be read 15 or 50 years later?

→ An old concern...

- Can we really do better with computers?

→ A computer museum? ☺



La pierre de Rosette (The Rosetta Stone).

What has not changed for 50 years in information representation?

What has not changed for 50 years in information representation?

The Data Exchange Problem

- Often, data must be sent to a third-party program/person
- Data must be made explicit (*e.g.* storage in files)
- Naïve[1] approach for defining a *file format*:
  - Define (binary?) representation for data + instructions, *e.g.* records
  - Write file format spec (v1.0?) + implement parser

[1] But used to be very widespread...

The Data Exchange Problem

- Often, data must be sent to a third-party program/person
- Data must be made explicit (*e.g.* storage in files)
- Naïve[1] approach for defining a *file format*:
  - Define (binary?) representation for data + instructions, *e.g.* records
  - Write file format spec (v1.0?) + implement parser

MachinCompany
File format X
Parser P_X

[1] But used to be very widespread...

The Data Exchange Problem

- Often, data must be sent to a third-party program/person
- Data must be made explicit (*e.g.* storage in files)
- Naïve[1] approach for defining a *file format*:
  - Define (binary?) representation for data + instructions, *e.g.* records
  - Write file format spec (v1.0?) + implement parser

MachinCompany
File format X
Parser P_X

Bidule Inc.
File format Y
Parser P_Y

[1] But used to be very widespread...

The Data Exchange Problem

- Often, data must be sent to a third-party program/person
- Data must be made explicit (*e.g.* storage in files)
- Naïve[1] approach for defining a *file format*:
  - Define (binary?) representation for data + instructions, *e.g.* records
  - Write file format spec (v1.0?) + implement parser

MachinCompany
File format X
Parser P_X

Bidule Inc.
File format Y
Parser P_Y

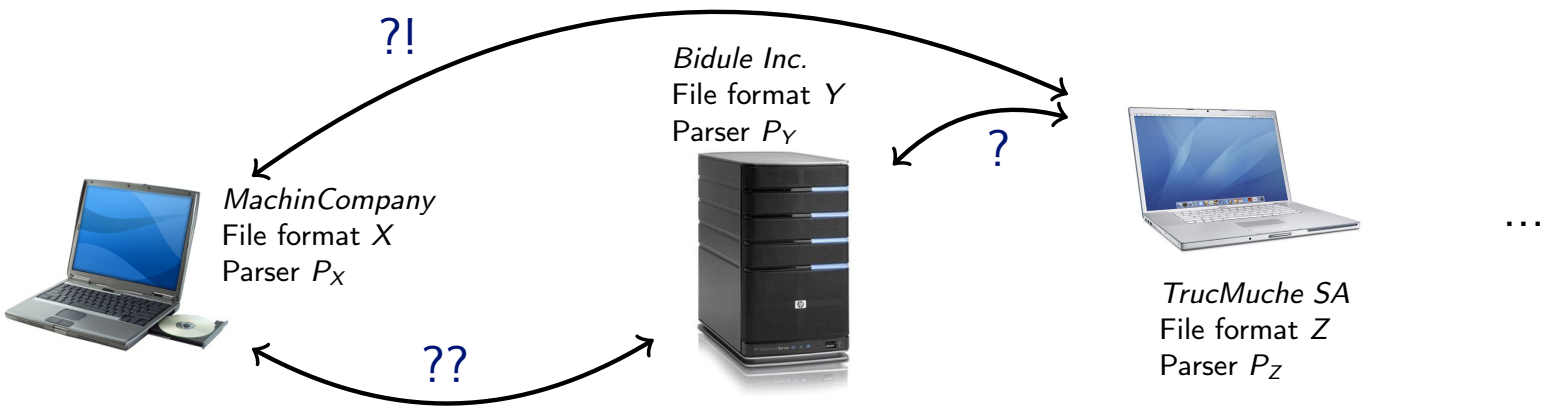
TrucMuche SA
File format Z
Parser P_Z

...

[1] But used to be very widespread...

The Data Exchange Problem

- Often, data must be sent to a third-party program/person
- Data must be made explicit (*e.g.* storage in files)
- Naïve[1] approach for defining a *file format*:
  - Define (binary?) representation for data + instructions, *e.g.* records
  - Write file format spec (v1.0?) + implement parser



- **Problems:** - exchanging data → exchanging programs!
  - this approach cannot scale (and costs $\$^{\$\$\$}$)

→ Need for normalization of data exchange

[1] But used to be very widespread...

Motivation for XML:

To have one language to describe and exchange data

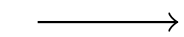
XML = Data

```
Pierre Genevès
CNRS
pierre.geneves@inria.fr

...

Nabil Layaïda
INRIA
nabil.layaida@inria.fr
```

Text file

XML = Data + Structure

Text file



"Mark it up!"

```
<people>
 <person>
  <name>Pierre Genevès</name>
  <affil>CNRS</affil>
  <email>pierre.geneves@inria.fr</email>
 </person>
 ...
 <person>
  <name>Nabil Layaïda</name>
  <affil>INRIA</affil>
  <email>nabil.layaida@inria.fr</email>
  </person>
</people>
```

XML Document

XML = Data + Structure

Text file

"Mark it up!"

```
<people>
 <person>
  <name>Pierre Genevès</name>
  <affil>CNRS</affil>
  <email>pierre.geneves@inria.fr</email>
 </person>
 ...
 <person>
  <name>Nabil Layaïda</name>
  <affil>INRIA</affil>
  <email>nabil.layaida@inria.fr</email>
  </person>
</people>
```

XML Document

Tags describe structure, independently from processors (tags are **not** implicit parameters for a given processor, *e.g.* tags are **not** intended for describing presentation)

XML = Data + Structure

Text file

"Mark it up!"

```
<people>
 <person>
  <name>Pierre Genevès</name>
  <affil>CNRS</affil>
  <email>pierre.geneves@inria.fr</email>
 </person>
 ...
 <person>
  <name>Nabil Layaïda</name>
  <affil>INRIA</affil>
  <email>nabil.layaida@inria.fr</email>
  </person>
</people>
```

XML Document

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

XML Documents

- Ordinary text files (UTF8, UTF16, ...)
- Originates from typesetting/DocProcessing community
- Idea of labeled brackets ("mark up") for structure is not new (already used by Chomsky in the 1960's)
- Properly nested brackets/tags describe a **tree structure**
- Allows applications from different vendors to exchange data
- **Standardized, extremely widely accepted**
- A Lingua franca for structured data exchange...

Standards for Data Exchange

- Before: file format tied to a processor (due to processor-specific instructions)
- After: markup language for describing (structured) data in itself (independently from processors)

XML History

Ancestors

1974 SGML (Charles Goldfarb at IBM Research)

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

1994 Berners-Lee founds World Wide Web Consortium (W3C)

1996 **XML** (W3C draft, v1.0 in 1998)
http://www.w3.org/TR/REC-xml/

Initial W3C Goals for XML[2]

"The design goals for XML are:

1. *XML shall be straightforwardly usable over the Internet.*
2. *XML shall support a wide variety of applications.*
3. *XML shall be compatible with SGML.*
4. *It shall be easy to write programs which process XML documents.*
5. *The number of optional features in XML is to be kept to the absolute minimum, ideally zero.*
6. *XML documents should be human-legible and reasonably clear.*
7. *The XML design should be prepared quickly.*
8. *The design of XML shall be formal and concise.*
9. *XML documents shall be easy to create.*
10. *Terseness is of minimal importance."*

[2] `http://www.w3.org/TR/WD-xml-961114`

XML is a Data Exchange Format

- **Contra..** extremely verbose, lots of repetitive markup, large files
- **Pro..** answers ambitious goals:
  - → long-standing (mark-up does not depend on the system where it was created nor on processings)
  - → One of the pillars of the web
  - → We have **A STANDARD!**
  - → If you use XML properly, you will never need to write a parser again

XML is a Meta-Language

- XML makes it possible to create Markup-Languages
- Instead of writing a parser, **you simply fix your own "XML Dialect"**

  by describing all "admissible structures" :

  - allowed element names
  - how they can be assembled together
  - maybe even the specific data types that may appear inside

  You do this using an XML Type definition language such as DTD[3], XML Schema (W3C), or Relax NG (Oasis).

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

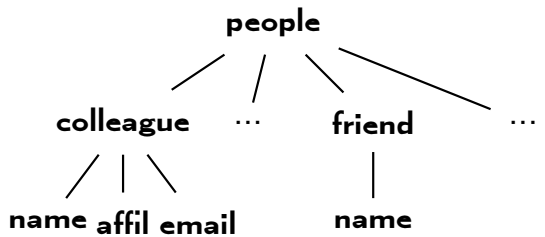
[3] Part of the XML Recommendation: http://www.w3.org/TR/REC-xml/16

XML Document Type Definition

- The XML Recommendation[4] includes an XML type definition language for specifying **document types**: DTD

people	$\rightarrow$	(colleague | friend)*
colleague	$\rightarrow$	name, affil$^+$, email
friend	$\rightarrow$	name, affil*, phone*, email?
...		

Document Type



- Each element is associated with its content model: a reg. expr. (, | ? * +)
- A document type (a set of such associations + a particular root element) describes a set of valid documents used by an organisation

[4] http://www.w3.org/TR/REC-xml/

XML: What Else?

- → Attributes
- → Comments
- → Processing Instructions
- → Entity References
- → Namespaces
- → Text (a specific node kind)

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

- → Attributes
- → Comments
- → Processing Instructions
- → Entity References
- → Namespaces
- → Text (a specific node kind)

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

```
<!-- a comment here -->
```

- → Attributes
- → **Comments**
- → Processing Instructions
- → Entity References
- → Namespaces
- → Text (a specific node kind)

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

```
<!-- a comment here -->
```

- → Attributes
- → **Comments**
- → Processing Instructions
- → Entity References
- → Namespaces
- → Text (a specific node kind)

```
<?php sql("SELECT * FROM .") ...  ?>
```

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

```
<!-- a comment here -->
```

- → Attributes
- → **Comments**
- → Processing Instructions
- → **Entity References**
- → Namespaces
- → Text (a specific node kind)

```
<?php sql("SELECT * FROM .") ...  ?>
```

DTD: `<!ENTITY notice "All rights...">`

instance: `<p> Copyright:  &notice; </p>`

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

```
<!-- a comment here -->
```

- → Attributes
- → **Comments**
- → Processing Instructions
- → **Entity References**
- → **Namespaces**
- → Text (a specific node kind)

```
<?php sql("SELECT * FROM .") ...  ?>
```

DTD: `<!ENTITY notice "All rights...">`

instance: `<p> Copyright:  &notice; </p>`

```
<book xmlns="http://www.books.com/xml"
      xmlns:cars="http://www.cars.com/xml">
 <part><cars:part>Avoids collisions..</cars:part></part>
</book>
```

XML: What Else?

```
<article class="story" id="news7" datetime="2017-08-08">..</article>
```

```
<!-- a comment here -->
```

- → Attributes
- → Comments
- → Processing Instructions
- → **Entity References**
- → **Namespaces**
- → **Text (a specific node kind)**

```
<?php sql("SELECT * FROM .") ...  ?>
```

DTD: `<!ENTITY notice "All rights...">`

instance: `<p> Copyright:  &notice; </p>`

```
<book xmlns="http://www.books.com/xml"
      xmlns:cars="http://www.cars.com/xml">
 <part><cars:part>Avoids collisions..</cars:part></part>
</book>
```

XML Today

"There is essentially no computer in the world, desktop, handheld, or backroom, that doesn't process XML sometimes..."

Some Widespread XML Dialects...

- XHTML (W3C) – the XML version of HTML
- SVG (W3C) – Animated Vector Graphics
- SMIL (W3C) – Synchronized Multimedia Documents, and MMS
- MathML (W3C) – Mathematical formulas
- XForms (W3C) – Web forms
- Fix, FPML – Financial structured products, transactions ...
- CML – Chemical molecules
- X3D (Web3D) - 3D Graphics
- XUL (Mozilla), MXML (Macromedia), XAML (Microsoft) – Interface Definition Languages
- SOAP (RPC using HTTP), WSDL (W3C), WADL (Sun) – Web Services
- RDF (W3C), OWL (W3C) – Metadata/Knowledge in the Semantic Web
- ...

Outline of the Sequel

- Two notions of correctness:
  - Well-formedness
  - Validity
- Defining your own classes of documents
  - DTDs, XML Schemas
  - Modeling trees and graphs
- Parsing (with or without validation)

XML Defines 2 Levels of Correctness

1. Well-formed XML (minimal requirement)
   - The **flat text format** seen on the **physical** side, *i.e.* a set of (UTF8/16) character sequences being well-formed XML
   - Ensures data correspond to logical tree-like structures (applications that want to analyse and transform XML data in any meaningful manner will find processing flat character sequences hard and inefficient)
2. Valid XML (optional, stricter requirement)
   - More often than not, applications require the XML input **trees** to conform to a **specific XML dialect**, defined by *e.g.* a DTD

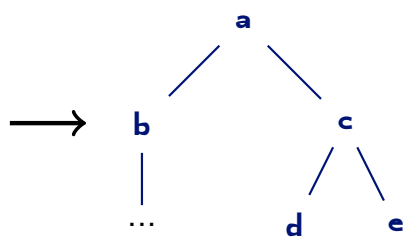
Well-Formed XML

```
<a att="v">
  <bar><e/></foo>
  <c>...</c>
</a>
```

```
<a att="v1" att="v2">
  <b>...</b>
  <c><e></c></e>
</a>
```

Not Well-formed XML

```
<a>
  <b>...</b>
  <c><d/><e/></c>
</a>
```



Well-formed XML

Characters	<	>	"	'	&
Entities	<	>	"	'	&

- Proper nesting of opening/closing tags
- Shortcut: <e/> for <e></e>
- Every attribute must have a (unique) value
- A document has one and only one root
- No ambiguity between structure and data

→ Any XML processor considers well-formed XML as a logical tree structure which is:

  - ordered (except attributes!)
  - finite (leaves are empty elements or character data)

→ It **must** stop for not well-formed XML.

Valid XML

- The header of a document **may** include a reference to a DTD:

  ```
  <!DOCTYPE root PUBLIC "public-identifier" "uri.dtd">
  ```

- A document with such a declaration must be valid wrt the declared type

→ The parser will validate it

Example

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
 <html>
   ...
 </html>
```

Why Validate?

- A document type is a contract between data producers and consumers
- Validation allows:
  - → a producer to check that he produces what he promised
  - → a consumer to check what the producer sends
  - → a consumer to protect his application
  - → **leaving error detection up to the parser**
  - → simplifying applications (we know where to find relevant information in valid documents)
  - → delivering high-speed XML throughput (once the input is validated, a lot of runtime checks can be avoided)

Zoom on Document Type Definition (DTD)

- Any element e to be used in the XML dialect needs to be introduced via

  `<!ELEMENT e` *cm*`>`

Content model *cm*	**Valid content**
`ANY`	arbitrary well-formed XML content
`EMPTY`	no child elements allowed (attributes OK)
Reg. exp. over tag names, `#PCDATA`, and constructors `| , + * ?`	order and occurence of child elements and text content must match the regular expression

- Example (XHTML 1.0 Strict DTD): `<!ELEMENT img EMPTY>`

Reg. Exp. in DTD Content Models

Reg. Exp.	**Semantics**
`tagname`	element named `tagname`
`#PCDATA`	text content (parsed character data)
c_1, c_2	c_1 directly followed by c_2
$c_1 \mid c_2$	c_1 or, alternatively, c_2
c^+	c, one or more times
c^*	c, zero or more times
$c?$	optional c

Example: recipes.xml (fragment)

```
1 <!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>
2 <!ELEMENT title (#PCDATA)>
3 <!ELEMENT ingredient (ingredient*,preparation)?>
4 <!ELEMENT preparation (step*)>
```

Declaring Attributes

- Using the DTD `ATTLIST` declaration, validation of XML documents is extended to attributes
- The `ATTLIST` declaration associates a list of attribute names a_i with their owning element `e`:

  `<!ATTLIST e`

  $a_1 \quad \tau_1 \quad d_1$

  ...

  $a_n \quad \tau_n \quad d_n$

  `>`

  $\rightarrow$ The attribute types τ_i define which values are valid for attributes a_i.

  $\rightarrow$ The defaults d_i indicate if a_i is required or optional (and, if absent, if a default value should be assumed for a_i).

  $\rightarrow$ In XML, attributes of an element are unordered. The `ATTLIST` declaration prescribes no order of attribute usage.

- Via attribute types, control over the valid attribute values can be exercised:

Attribute Type τ_i	**Semantics**
`CDATA`	character data (no < but `<`, ...)
$(v_1 \mid v_2 \mid \ldots \mid v_n)$	enumerated literal values
`ID`	value is document-wide unique identifier for owner element
`IDREF`	references an element via its `ID` attribute

Example: academic.xml

```
1 <!ELEMENT academic (Firstname, Middlename*, Lastname)>
2 <!ATTLIST academic
3    title (Prof|Dr) #REQUIRED
4    team CDATA #IMPLIED
5 >
6
7  <academic title="Dr" team="Tyrex"> ... </academic>
```

- Attribute defaulting in DTDs:

Attribute Default d_i	**Semantics**
`#REQUIRED`	element must have attribute a_i
`#IMPLIED`	attribute a_i is optional
v (a value)	attribute a_i is optinal, if absent, default value v for a_i is assumed
`#FIXED` v	attribute a_i is optional, if present, must have value v

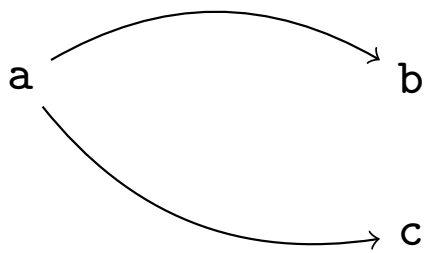
Example: contacts.xml

```
1 <!ELEMENT contact (name, email+, phone*)>
2 <!ATTLIST contact
3    emailMode (text|xhtml) "text"     <!--send safely-->
4 >
```

Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data
- Attributes of type `ID` and `IDREF` may be used to encode graph structures in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML nodes `a` and `b`:

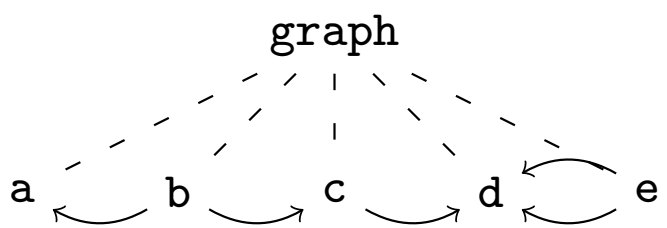
1. attach a unique identifier to node `b` (using an `ID` attribute), then
2. refer to `b` from `a` via this identifier (using an `IDREF` attribute).
3. For an outdegree > 1 (see below), use an `IDREFS` attribute.



Graphs in XML – An Example

Graph.xml

```
1  <?xml version="1.0"?>
2  <!DOCTYPE graph [
3    <!ELEMENT graph (node+) >
4    <!ELEMENT node ANY > <!-- attach arbitrary data to a node -->
5    <!ATTLIST node
6      id ID #REQUIRED
7      edges IDREFS #IMPLIED > <!-- we may have nodes with outdegree 0 -->
8  ]>
9
10 <graph>
11   <node id="A">a</node>
12   <node id="B" edges="A C">b</node>
13   <node id="C" edges="D">c</node>
14   <node id="D">d</node>
15   <node id="E" edges="D D">e</node>
16 </graph>
```



Example (Character references in "ComicsML")

ComicsML.dtd (fragment)

```
1  <!DOCTYPE strip [
2    ...
3    <!ELEMENT character (#PCDATA) >
4    <!ATTLIST character
5        id        ID                          #REQUIRED >
6    <!ELEMENT bubble (#PCDATA) >
7    <!ATTLIST bubble
8        speaker   IDREF                       #REQUIRED
9        to        IDREFS                      #IMPLIED
10       tone      (angry|question|...)        #IMPLIED >
11 ]>
```

Validation results (message generated by Apache's Xerces):

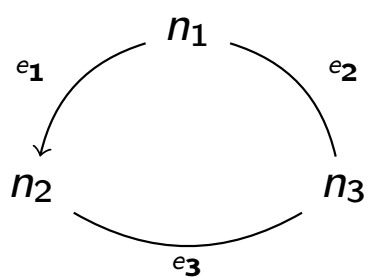
- Setting attribute to some random non-existent character identifier:
  `ID attribute 'yoda' was referenced but never declared`
- Using a non-enumerated value for attribute `tone`:
  `Attribute 'tone' does not match its defined enumeration list`

A Real-world DTD – GraphML

- GraphML[5] has been designed to provide a convenient file format to represent arbitrary graphs
- Graphs (element `graph`) are specified as lists of `nodes` and `edges`
- Edges point from `source` to `target`
- Nodes and edges may be annotated using arbitrary description and data
- Edges may be `directed` (+ attribute `edgedefault` of `graph`)

Graph.xml

```
1  <graphml>
2    <graph edgedefault="undirected">
3      <node id="n1"/>
4      <node id="n2"/>
5      <node id="n3"/>
6      <edge id="e1" source="n1" target="n2" directed="true"/>
7      <edge id="e2" source="n2" target="n3" directed="false"/>
8      <edge id="e3" source="n3" target="n1"/>
9    </graph>
10 </graphml>
```



[5] http://graphml.graphdrawing.org/

GraphML.dtd (main part)

```
<!-- GRAPHML DTD (flat version) ========================================== -->
<!ELEMENT graphml  ((desc)?,(key)*,((data)|(graph))*)>

<!ELEMENT locator EMPTY>
<!ATTLIST locator
          xmlns:xlink   CDATA    #FIXED    "http://www.w3.org/TR/2000/PR-xlink-20001220/"
          xlink:href    CDATA    #REQUIRED
          xlink:type    (simple) #FIXED    "simple">

<!ELEMENT desc (#PCDATA)>

<!ELEMENT graph    ((desc)?,((((data)|(node)|(edge)|(hyperedge))*)|(locator)))>
<!ATTLIST graph
          id          ID                     #IMPLIED
          edgedefault (directed|undirected) #REQUIRED>

<!ELEMENT node   (desc?,(((data|port)*,graph?)|locator))>
<!ATTLIST node
          id        ID      #REQUIRED>

<!ELEMENT port ((desc)?,((data)|(port))*)>
<!ATTLIST port
          name    NMTOKEN  #REQUIRED>

<!ELEMENT edge ((desc)?,(data)*,(graph)?)>
<!ATTLIST edge
          id         ID           #IMPLIED
          source     IDREF        #REQUIRED
          sourceport NMTOKEN      #IMPLIED
          target     IDREF        #REQUIRED
          targetport NMTOKEN      #IMPLIED
          directed   (true|false) #IMPLIED>

<!ELEMENT key (#PCDATA)>
<!ATTLIST key
          id  ID                                              #REQUIRED
          for (graph|node|edge|hyperedge|port|endpoint|all) "all">

<!ELEMENT data  (#PCDATA)>
<!ATTLIST data
          key      IDREF        #REQUIRED
          id       ID           #IMPLIED>
```

Concluding Remarks

- DTD syntax:
  - ✓ **Pro:** compact, easy to understand
  - × **Con:** ?

Concluding Remarks

- DTD syntax:
  - ✓ **Pro:** compact, easy to understand
  - × **Con:** not in XML!
- DTD functionality:
  - × no fine-grained types (everything is character data; what about, *e.g.* integers?)
  - × no further occurence constraints (*e.g.* cardinality of sequences)

→ DTD is a very simple but quite limited type definition language

XML Schema

- With XML Schema[6], W3C provides an XML type definition language that goes beyond the capabilities of the "native" DTD concept:
  - XML Schema descriptions are valid XML documents themselves
  - XML Schema provides a rich set of built-in data types
  - Users can extend this type system via user-defined types
  - XML element (and attribute) types may even be derived by inheritance

XML Schema *vs.* DTDs

→ Why would you consider its XML syntax as an advantage?

[6] http://www.w3.org/TR/xmlschema-0/

Some XML Schema Constructs

- Declaring an element

```
<xsd:element name="author"/>
```

No further typing specified: the `author` element may contain string values only.

- Declaring an element with bounded occurence

```
<xsd:element name="character" minOccurs="0" maxOccurs="unbounded"/>
```

Absence of `minOccurs/maxOccurs` implies exactly once.

- Declaring a typed element

```
<xsd:element name="year" type="xsd:date"/>
```

Content of year takes the format YYYY-MM-DD. Other simple types: `string`, `boolean`, `number`, `float`, `duration`, `time`, `AnyURI`, `...`

- Simple types are considered **atomic** with respect to XML Schema (e.g., the YYYY part of an xsd:date value has to be extracted by the XML application itself).

- Non-atomic complex types are built from simple types using type constructors.

Declaring sequenced content

```
1  <xsd:complexType name="Characters">
2    <xsd:sequence>
3      <xsd:element name="character" minOccurs="1"
4                                    maxOccurs="unbounded"/>
5    </xsd:sequence>
6  </xsd:complexType>
7  <xsd:complexType name="Prolog">
8    <xsd:sequence>
9      <xsd:element name="series"/>
10     <xsd:element name="author"/>
11     <xsd:element name="characters" type="Characters"/>
12   </xsd:sequence>
13 </xsd:complexType>
14 <xsd:element name="prolog" type="Prolog"/>
```

An `xsd:complexType` may be used anonymously (no `name` attribute).

- With attribute `mixed="true"`, an `xsd:complexType` admits **mixed content**.

- New complex types may be **derived** from an existing (base) type.

Deriving a new complex type

```
<xsd:element name="newprolog">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Prolog">
        <xsd:element name="colored" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

- **Attributes** are declared within their owner element.

Declaring attributes

```
<xsd:element name="strip">
  <xsd:attribute name="copyright"/>
  <xsd:attribute name="year" type="xsd:gYear"/> ...
</xsd:element>
```

Other `xsd:attribute` modifiers: `use` (`required`, `optional`, `prohibited`), `fixed`, `default`.

- The validation of an XML document against an XML Schema goes as far as peeking into the **lexical representation** of simple typed values.

Restricting the value space of a simple type (enumeration)

```
1 <xsd:simpleType name="Car">
2   <xsd:restriction base="xsd:string">
3     <xsd:enumeration value="Audi"/>
4     <xsd:enumeration value="BMW"/>
5     <xsd:enumeration value="VW"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

Restricting the value space of a simple type (regular expression)

```
1 <xsd:simpleType name="AreaCode">
2   <xsd:restriction base="xsd:string">
3     <xsd:pattern value="0[0-9]+"/>
4     <xsd:minLength value="3"/>
5     <xsd:maxLength value="5"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

- Other **facets**: `length`, `maxInclusive` (upper bound for numeric values)...

Other XML Schema Concepts

- **Fixed** and **default** element content,
- support for **null values**,
- uniqueness constraints, arbitrary **keys** (specified via XPath)

Intermediate Outline

- Motivations and key principles behind XML
- Languages for defining sets of documents (tree languages)
- Processing XML Documents
- Parsing
  - Two radically different approaches: DOM and SAX
  - Advantages and drawbacks

XML Processing Model

Validation is good

- Validation is better than writing code
- Remember the promise:

  "**you will never have to write a parser again**"
  $\rightarrow$ instead, you will need to encode a grammar...
- Virtually all XML applications operate on the logical tree view which is provided to them by an XML parser
- An XML parser can be validating or non-validating
- XML parsers are widely available (*e.g.* Apache's Xerces).
- How is the XML parser supposed to communicate the XML tree structure to the application?

XML Parsers

- Two different approaches:
  1. Parser stores document into a fixed (standard) data structure (*e.g.* DOM)

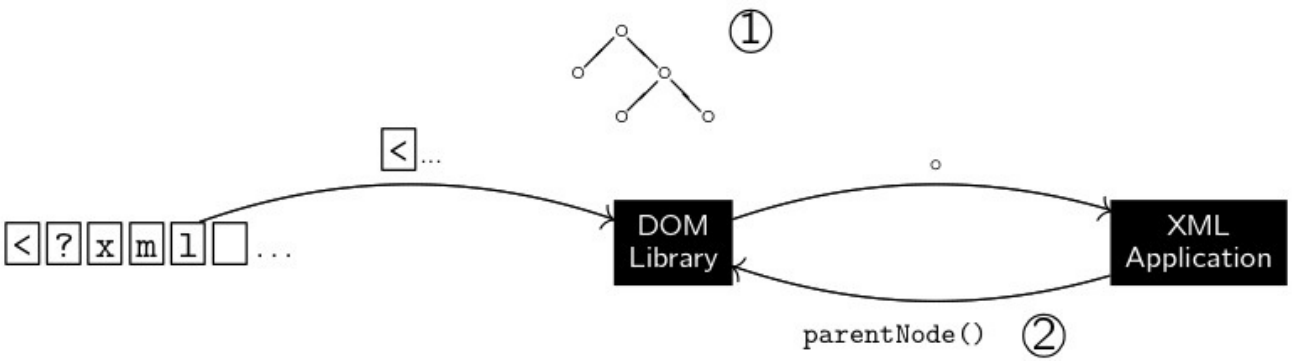
```
parser.parse("foo.xml");
doc = parser.getDocument();
```

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

Next slides on DOM & SAX by Marc H. Scholl (Uni KN)...

DOM—Document Object Model

- With **DOM**, W3C has defined a **language-** and **platform-neutral** view of XML documents.
- DOM APIs exist for a wide variety of—predominantly object-oriented—programming languages (Java, C++, C, Perl, Python, . . .).
- The DOM design rests on two major concepts:
  1. An **XML Processor** offering a DOM interface parses the XML input document, and constructs the **complete XML document tree** (in-memory).
  2. The **XML application** then issues DOM library calls to **explore** and **manipulate** the XML document, or **generate** new XML documents.



- (The complete DOM interface is too large to list here.) Some methods of the principal DOM types *Node* and *Document*:

DOM Type	Method		Comment
Node	*nodeName*	:: *DOMString*	redefined in subclasses, *e.g.*, tag name for *Element*, `"#text"` for *Text* nodes, ...
	parentNode	:: *Node*	
	firstChild	:: *Node*	leftmost child node
	nextSibling	:: *Node*	returns `NULL` for root element or last child or attributes
	childNodes	:: *NodeList*	see below
	attributes	:: *NameNodeMap*	see below
	ownerDocument	:: *Document*	
	replaceChild	:: *Node*	replace new for old node, returns old
Document	*createElement*	:: *Element*	creates element with given tag name
	createComment	:: *Comment*	creates comment with given content
	getElementsByTagName	:: *NodeList*	list of all *Elem* nodes in document order

Example: C++/DOM Code

content.cc (1)

```
1  // Xerces C++ DOM API support
2  #include <dom/DOM.hpp>
3  #include <parsers/DOMParser.hpp>
4
5  void collect (DOM_NodeList ns)
6  {
7    DOM_Node n;
8
9    for ( unsigned long i = 0;
10         i < ns.getLength ();
11         i++){
12     n = ns.item (i);
13
14     switch (n.getNodeType ()) {
15     case DOM_Node::TEXT_NODE:
16       cout << n.getNodeValue ().transcode ();
17       break;
18     case DOM_Node::ELEMENT_NODE:
19       collect (n.getChildNodes ());
20     }
21   }
22 }
```

content.cc (2)

```
23
24 void content (DOM_Document d)
25 {
26   collect (d.getChildNodes ());
27 }
28
29 int main (void)
30 {
31   XMLPlatformUtils::Initialize ();
32
33   DOMParser parser;
34   DOM_Document doc;
35
36   parser.parse ("foo.xml");
37   doc = parser.getDocument ();
38
39   content (doc);
40
41   return 0;
42 }
```

Now: Find all occurrences of `Dogbert` speaking (attribute `speaker` of element `bubble`) ...

dogbert.cc (1)

```
1  // Xerces C++ DOM API support
2  #include <dom/DOM.hpp>
3  #include <parsers/DOMParser.hpp>
4
5  void dogbert (DOM_Document d)
6  {
7    DOM_NodeList     bubbles;
8    DOM_Node         bubble, speaker;
9    DOM_NamedNodeMap attrs;
10
11   bubbles = d.getElementsByTagName ("bubble");
12
13   for (unsigned long i = 0; i < bubbles.getLength (); i++) {
14     bubble = bubbles.item (i);
15
16     attrs = bubble.getAttributes ();
17     if (attrs != 0)
18       if ((speaker = attrs.getNamedItem ("speaker")) != 0)
19         if (speaker.getNodeValue ().
20               compareString (DOMString ("Dogbert")) == 0)
21           cout << "Found Dogbert speaking." << endl;
22   }
23 }
```

dogbert.cc (2)

```
24
25  int main (void)
26  {
27    XMLPlatformUtils::Initialize ();
28
29    DOMParser parser;
30    DOM_Document doc;
31
32    parser.parse ("foo.xml");
33    doc = parser.getDocument ();
34
35    dogbert (doc);
36
37    return 0;
38  }
```

DOM—A Memory Bottleneck

- The two-step processing approach (① parse and construct XML tree, ② respond to DOM property function calls) enables the DOM to be "**random access**":

  The XML application may explore and update any portion of the XML tree at any time.
- The inherent memory hunger of the DOM may lead to
  1. heavy **swapping** activity
     (partly due to unpredictable memory access patterns, `madvise()` less helpful)
     or
  2. even "`out-of-memory`" failures.
     (The application has to be extremely careful with its own memory management, the very least.)

Numbers

DOM and random node access

Even if the application touches a single element node only, the DOM API has to maintain a data structure that represents the **whole XML input document** (all sizes in kB):[6]

XML size	**DOM process size** `DSIZ`	$\frac{\texttt{DSIZ}}{\text{XML size}}$	**Comment**
7480	47476	6.3	(Shakespeare's works) many elements containing small text fragments
113904	552104	4.8	(Synthetic eBay data) elements containing relatively large text fragments

[6] The random access nature of the DOM makes it hard to provide a truly "lazy" API implementation.

To remedy the memory hunger of DOM-based processing . . .

- Try to **preprocess** (*i.e.*, filter) the input XML document to reduce its overall size.
  - Use an XPath/XSLT processor to preselect *interesting* document regions,
  -  *no updates* to the input XML document are possible then,
  -  make sure the XPath/XSLT processor is *not* implemented on top of the DOM.

Or

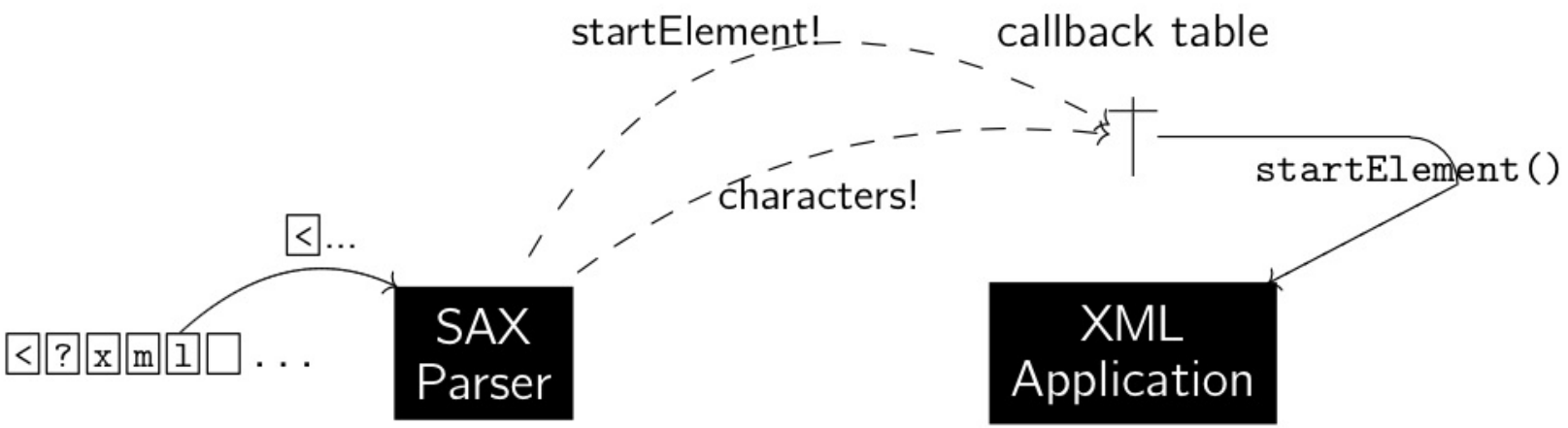
- Use a **completely different** approach to XML processing ($\rightarrow$ **SAX**).

SAX—Simple API for XML

- **SAX**[7] (**Simple API for XML**) is, unlike DOM, *not* a W3C standard, but has been developed jointly by members of the `XML-DEV` mailing list (*ca.* 1998).
- SAX processors use **constant space**, regardless of the XML input document size.
  - Communication between the SAX processor and the backend XML application does *not* involve an intermediate tree data structure.
  - Instead, the **SAX parser sends events** to the application whenever a certain piece of XML text has been recognized (*i.e.*, parsed).
  - The **backend acts on/ignores events** by populating a **callback function table**.

[7] `http://www.saxproject.org/`

Sketch of SAX's mode of operations



- A SAX processor reads its input document **sequentially** and **once** only.
- No memory of what the parser has seen so far is retained while parsing. As soon as a  *significant bit of XML text* has been recognized, an **event** is sent.
- The application is able to act on events **in parallel** with the parsing progress.

SAX Events

- To meet the constant memory space requirement, SAX reports **fine-grained parsing events** for a document:

Event	**...reported when seen**	**Parameters sent**
startDocument	<?xml...?>[8]	
endDocument	⟨EOF⟩	
startElement	<t a_1=v_1 ... a_n=v_n>	t, $(a_1, v_1), \ldots, (a_n, v_n)$
endElement	</t>	t
characters	*text content*	Unicode buffer ptr, length
comment	<!--c-->	c
processingInstruction	<?t pi?>	t, pi
	⋮	

[8] **N.B.**: Event *startDocument* is sent even if the optional XML text declaration should be missing.

dilbert.xml

```
1  <?xml encoding="utf-8"?> ⋆1
2  <bubbles> ⋆2
3    <!-- Dilbert looks stunned --> ⋆3
4    <bubble speaker="phb" to="dilbert"> ⋆4
5      Tell the truth, but do it in your usual engineering way
6      so that no one understands you. ⋆5
7    </bubble> ⋆6
8  </bubbles> ⋆7 ⋆8
```

Event[9] [10]		Parameters sent
$\star_1$	*startDocument*	
$\star_2$	*startElement*	t = "bubbles"
$\star_3$	*comment*	c = "␣Dilbert looks stunned␣"
$\star_4$	*startElement*	t = "bubble", ("speaker","phb"), ("to","dilbert")
$\star_5$	*characters*	*buf* = "Tell the...understands you.", *len* = 99
$\star_6$	*endElement*	t = "bubble"
$\star_7$	*endElement*	t = "bubbles"
$\star_8$	*endDocument*	

[9] Events are reported in **document reading order** $\star_1, \star_2, \ldots, \star_8$.

[10] **N.B.**: Some events suppressed (white space).

SAX Callbacks

- To provide an efficient and tight **coupling** between the SAX **frontend** and the application **backend**, the SAX API employs **function callbacks**:[11]
  1. Before parsing starts, the application **registers function references** in a table in which each event has its own slot:

Event	*Callback*
⋮	
startElement	?
endElement	?
⋮	

$\longrightarrow$ *SAX register*(*startElement*, `startElement ()`)
SAX register(*endElement*, `endElement ()`)

Event	*Callback*
⋮	
startElement	`startElement ()`
endElement	`endElement ()`
⋮	

  2. The application alone decides on the implementation of the functions it registers with the SAX parser.
  3. **Reporting an event** $\star_i$ then amounts to call the function (with parameters) registered in the appropriate table slot.

[11] Much like in event-based GUI libraries.

Java SAX API

In Java, populating the callback table is done via implementation of the SAX `ContentHandler` interface: a `ContentHandler` object represents the callback table, its methods (*e.g.*, `public void endDocument ()`) represent the table slots.

Example: Reimplement *content.cc* shown earlier for DOM (find all XML text nodes and print their content) using SAX (pseudo code):

content (*File f*)

// register the callback,
// we ignore all other events
SAX register (*characters*, *printText*);
SAX parse (*f*);
return;

printText ((*Unicode*) *buf*, *Int len*)

Int i;

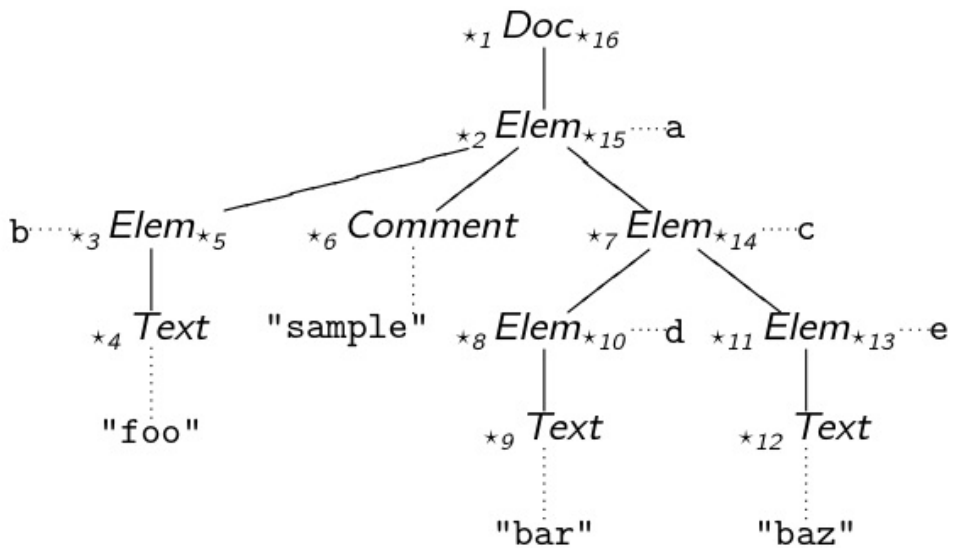
foreach $i \in 1 \ldots len$ **do**
  print (*buf*[*i*]);
return;

SAX and the XML Tree Structure

- Looking closer, the **order** of SAX events reported for a document is determined by a **preorder traversal** of its document tree[12]:

Sample XML document

```
1  ⋆1
2  <a>⋆2
3     <b>⋆3 foo⋆4 </b>⋆5
4     <!--sample-->⋆6
5     <c>⋆7
6       <d>⋆8 bar⋆9 </d>⋆10
7       <e>⋆11 baz⋆12 </e>⋆13
8     </c>⋆14
9  </a>⋆15 ⋆16
```



N.B.: An *Elem* [*Doc*] node is associated with two SAX events, namely *startElement* and *endElement* [*startDocument*, *endDocument*].

[12] Sequences of sibling *Char* nodes have been collapsed into a single *Text* node.

Challenge

- This **left-first depth-first** order of SAX events is well-defined, but appears to make it hard to answer certain queries about an XML document tree.

✎ Collect all direct children nodes of an *Elem* node.

In the example on the previous slide, suppose your application has just received the *startElement*($t =$ `"a"`) event $\star_2$ (*i.e.*, the parser has just parsed the opening element tag <a>).

With the remaining events $\star_3 \ldots \star_{16}$ still to arrive, can your code detect all the immediate children of *Elem* node `a` (*i.e.*, *Elem* nodes `b` and `c` as well as the *Comment* node)?

The previous question can be answered more generally:

> *SAX events are sufficient to* **rebuild the complete XML document tree** *inside the application. (Even if we most likely don't want to.)*

SAX-based tree rebuilding strategy (sketch):

1. [*startDocument*]
   Initialize a **stack** S of **node IDs** (*e.g.* $\in \mathbb{Z}$). **Push** first ID for this node.
2. [*startElement*]
   Assign a **new ID** for this node. **Push** the ID onto S.[13]
3. [*characters*, *comment*, ...]
   Simply assign a new node ID.
4. [*endElement*, *endDocument*]
   Pop S (no new node created).

Invariant: The **top of** S holds the identifier of the current **parent node**.

[13] In callbacks ② and ③ we might wish to store further node details in a table or similar summary data structure.

Final Remarks on SAX

- For an XML document fragment shown on the left, SAX might actually report the events indicated on the right:

XML fragment

```
1  <affiliation>
2    AT&amp;T Labs
3  </affiliation>
```

XML + SAX events

```
1  <affiliation>⋆1
2    AT⋆2&amp;⋆3T Labs
3  ⋆4</affiliation>⋆5
```

$\star_1$	*startElement*(affiliation)
$\star_2$	*characters*("\n␣␣AT", 5)
$\star_3$	*characters*("&", 1)
$\star_4$	*characters*("T␣Labs\n", 7)
$\star_5$	*endElement*(affiliation)

White space is reported.

Multiple *characters* **events** may be sent for text content (although adjacent).

(Often SAX parsers break text on entities, but may even report each character on its own.)

Concluding Remarks

We have seen:

- Motivation for XML (where XML originates from and what it is aimed for)
- What is fundamental with XML:
  1. standard for structured information
  2. independence from processors
  3. well-formed documents can be processed as trees
  4. users may agree on a dialect and save coding effort, they can exchange valid documents and the schemas...
- How the XML meta-language works, how to define your own XML dialect (using a schema language e.g. DTD, XML Schema...)
- How/when to use the 2 different kinds of XML parsers (DOM, SAX)

$\rightarrow$ Welcome to the world of tree-structured information!