What is XML?

- Extensible Markup Language is not a markup language!
  - does not fix a tag set nor its semantics
    (unlike markup languages, e.g. HTML)
- XML documents have no inherent (processing or presentation) semantics
  - even though many think that XML is semantic or self-describing; See next

What is XML (2)?

- XML is
  - a way to use markup to represent information
  - a metalanguage
    - supports definition of specific markup languages through XML DTDs or Schemas
    - e.g. XHTML is a reformulation of HTML using XML
- Often 'XML' = XML + XML technology
  - that is, processing models and languages we’re studying (and many others ...)

Semantics of XML Markup

- Meaning of this XML fragment?
  <student>
  <name>Pekka</name>
  <email>pekkeki@cs.oulu.fi</email>
  </student>

How does it look?

- XML documents are made of ISO-10646 (32-bit) characters; in practice of their 16-bit Unicode subset (used, e.g., in Java)
- Three aspects or characters:
  - identification as numeric code points
  - representation by bytes
  - visual presentation
External Aspects of Characters

- Documents are stored/transmitted as a sequence of bytes (of 8 bits). An encoding determines how characters are represented by bytes.
  - UTF-8 (≈ 7-bit ASCII) is the XML default encoding
  - encoding="iso-8859-1" -> 256 Western European chars as single bytes
- A font (collection of character images called glyphs) determines the visual presentation of characters

XML Encoding of Structure 1

- XML is, essentially, just a textual encoding scheme of labelled, ordered and attributed trees, in which
  - internal nodes are elements labelled by type names
  - leaves are text nodes labelled by string values, or empty element nodes
  - the left-to-right order of children of a node matters
  - element nodes may carry attributes (= name-string-value pairs)

XML Encoding of Structure 2

- XML encoding of a tree
  - corresponds to a pre-order walk
  - start of an element node with type name A denoted by a start tag <A>, and its end denoted by end tag </A>
  - possible attributes written within the start tag:
    - A attr="" value="" ... attr_h="" value_h=""
    - names must be unique: attr_f=attr_h when k ≠ h
    - text nodes written as their string value

XML: Logical Document Structure

- Elements
  - indicated by matching (case-sensitive!) tags
    - <ElementTypeName> ... </ElementTypeName>
  - can contain text and/or subelements
  - can be empty:
    - <elem-type></elem-type> or
    - <elem-type/> (e.g. <br/> in XHTML)
  - unique root element -> document a single tree

CDATA Sections

- "CDATA Sections" to include XML markup characters as textual content

```xml
<![CDATA[
Here we could include for example code fragments:
<example>if (Count < 5 && Count > 0) </example>
]]>
```

Logical document structure (2)

- Attributes
  - name-value pairs attached to elements
    - "metadata", usually not treated as content
  - in start-tag after the element type name
    - <div class="preface" date='990126'> ...
- Also:
  - <!-- comments outside other markup -->
  - <note this would be passed to the application as a processing instruction named 'note'>

Two levels of correctness (1)

- Well-formed documents
  - roughly: follows the syntax of XML, markup correct (elements properly nested, tag names match, attributes of an element have unique names, ...)
  - violation is a fatal error
- Valid documents
  - (in addition to being well-formed) obey an associated grammar (DTD/Schema)
2.2 Basics of document grammars

- Regular expressions describe regular languages:
  - relatively simple sets of strings over a finite alphabet of
  » characters, events, document elements, …
- Used as:
  - text searching patterns (e.g., emacs, ed, awk, grep, Perl)
  - part of grammatical formalisms (for programming languages, XML, in XML DTDs and schemas, …)
- Let us describe accepted contents of document components

Regular Expressions: Syntax

A regular expression (säännöllinen lauseke) over an alphabet \( \Sigma \) is either
- \( \emptyset \) an empty set,
- \( \lambda \) lambda (sometimes epsilon \( \epsilon \)),
- \( a \) any alphabet symbol \( a \in \Sigma \),
- \( (R \mid S) \) choice,
- \( (R \ast) \) concatenation, or
- \( (R) \ast \) Kleene closure or iteration

Where \( R \) and \( S \) are regular expressions

N.B.: Different syntaxes exist but the idea is same

Regular Expressions: Grouping

Conventions to simplify operator expressions by eliminating extra parentheses:
- outermost parentheses may be eliminated:
  \[ (L) \ast = E \]
- binary operations are associative:
  \[ (A \mid (B \mid C)) = (A \mid B) \mid C \]
  \[ (A \mid B) \mid C = (A \mid B) \mid C \]
- operations have priorities:
  iteration first, concatenation next, choice last
- for example:
  \[ (A \mid (B \ast C^*)) = A \mid (B C^*) \]

Regular Expressions: Examples

- Simplified top-level structure of a document:
  \( \Sigma = \{ \text{title, auth, date, sect} \} \)
  - title followed by an optional list of authors,
  - fby an optional date, fby one or more sections:
    title auth\ast sect sect\ast
  - Commonly used abbreviations:
    \( E^? = (E \mid \lambda) \), \( E^+ = E \ast E \ast \)
  - Followed by an optional date, fby one or more sections:
    title auth\ast date sect \ast
Context-Free Grammars (CFGs)

- Used widely to syntax specification (programming languages, XML, ...) and to parser/compiler generation (e.g. YACC/GNU Bison)
- CFG G formally a 4-tuple \( \langle \text{V}, \Sigma, P, S \rangle \)
  - \( V \) is the alphabet of the grammar \( G \)
  - \( \Sigma \subset V \) is a set of terminal symbols (päätesymbolit)
  - \( N = V - \Sigma \) is a set of nonterminal symbols (välikesymbolit)
  - \( P \) set of productions
  - \( S \in V \) the start symbol (lähtösymboli)

Productions and Derivations

- Productions: \( A \rightarrow \alpha, \text{where } A \in N, \alpha \in V^* \)
  - E.g.: \( A \rightarrow aBa \) (production 1)
- Let \( \gamma, \delta \in V^* \). String \( \gamma \) derives \( \delta \) directly if
  - \( \gamma = \alpha \beta \), \( \delta = \omega \beta \) for some \( \alpha, \beta \in V^* \), and \( A \rightarrow \omega \in P \)
  - E.g. \( AA \rightarrow AaBa \) (assuming production 1 above)
- NB: CFGs are often given simply by listing the productions \( P \)

Language Generated by a CFG

- \( \gamma \) derives \( \delta \), \( \gamma \Rightarrow \ast \delta \), if there’s a sequence of (0 or more) direct derivations that transforms \( \gamma \) to \( \delta \)
- The language generated by a CFG \( G \):
  - \( L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \} \)
- NB: \( L(G) \) is a set of strings;
  - To model document structures, we consider syntax trees (rakennepuu)

Syntax Trees

- A.k.a parse trees (jäsennyspuu) or derivation trees (johtopuu)
- Child nodes are ordered left-to-right
- Nodes are labelled by symbols of \( V \):
  - internal nodes by nonterminals, root by start symbol
  - leaves by terminal symbols (or empty string \( \lambda \))
- A node labelled \( A \) can have children labelled by \( X_1, \ldots, X_n \) only if \( A \rightarrow X_1, \ldots, X_n \in P \)

Syntax Trees: Example

- CFG for simplified arithmetic expressions:
  - \( V = \{ E, +, \ast, I \}; \Sigma = \{ +, \ast, I \}; N = \{ E \}; S = E \)
  - \( P = \{ E \rightarrow E+E, E \rightarrow E*E, E \rightarrow I, E \rightarrow (E) \} \)
- Syntax tree for \( 2*(3+4) \)?

Syntax Trees: Example

- Productions:
  - \( E \rightarrow E+E \)
  - \( E \rightarrow E*E \)
  - \( E \rightarrow I \)
  - \( E \rightarrow (E) \)

CFGs for Document Structures

- Nonterminals represent document elements
  - E.g. model for items (Ref) of a bibliography list:
    - \( \text{Ref} \rightarrow \text{AuthorList Title PublData} \)
    - \( \text{AuthorList} \rightarrow \text{Author AuthorList} \)
    - \( \text{AuthorList} \rightarrow \lambda \)
- Notice:
  - right-hand-side of a CFG production is a fixed string of grammar symbols
  - Repetition simulated using recursion
    - e.g. \( \text{AuthorList} \) above

Example: List of Three Authors

- \( \text{Ref} \rightarrow \text{AuthorList Title PublData} \)
- \( \text{AuthorList} \rightarrow \text{Author AuthorList} \)
- \( \text{AuthorList} \rightarrow \lambda \)
Problems

- "Auxiliary nonterminals" (like AuthorList) obscure the model
  - the last Author is several levels apart from its intuitive parent element Ref
  - clumsy to access and to count Authors of a reference
  - avoided by extended context-free grammars

Extended CFGs (ECFGs)

- like CFGs, but right-hand-sides of productions are regular expressions over V
  - E.g. Ref -> Author* Title PublData
- Let γ, δ ∈ V*. String γ derives δ directly, γ => δ, if
  - γ = αAβ, δ = αωβ for some α, β ∈ V*, and A -> E ∈ P such that ω ∈ L(E)
- E.g. Ref => Author Author Title PublData

Language Generated by an ECFG

- L(G) defined similarly to CFGs:
  - γ derives δ, γ => δ, if
    γ => α1 => ... => αn = δ
    L(G) = {w ∈ Σ* | S =>* w }
- Theorem: Extended and ordinary CFGs allow to generate the same (string) languages.

But syntax trees of ECFGs and CFGs differ! (Next)

Example: Three Authors of a Ref

- In (extended) CFGs:
  - Leaves of parse trees are labelled by single terminal symbols (∈ Σ)
  - Too granular for practise; instead terminal symbols which stand for all values of a type
  - XML DTDs: #PCDATA for variable length string content
- In XML Schema:
  - Explicit string literals rare in document grammars

Terminal Symbols in Practise

- Syntax Trees of an ECFG

- Problems
  - Theorem: Extended and ordinary CFGs allow to generate the same (string) languages.

Example: Three Authors of a Ref

- Ref -> Author* Title PublData ∈ P.
  - Author Author Author Title PublData ∈ L(Author* Title PublData)

Terminals

- Syntax Trees of an ECFG

- Problems
  - Theorem: Extended and ordinary CFGs allow to generate the same (string) languages.

Example: Three Authors of a Ref

- Ref -> Author* Title PublData ∈ P.
  - Author Author Author Title PublData ∈ L(Author* Title PublData)

No terminal symbols are used in an internal node's children (e.g., Authors below a Ref node)

2. 3 Basics of XML DTDs

- Syntax (in the prolog of a document instance):
  - DTD is union of the external and internal subset; internal has higher precedence
  - can override entity and attribute declarations (see next)
## Markup Declarations

- **DTD consists of markup declarations**
  - element type declarations
    - similar to productions of ECFGs
  - attribute-list declarations
    - for declared element types
  - entity declarations (see later)
  - notation declarations
    - to pass information about external (binary) objects

## How do Declarations Look Like?

- **Declaring elements**
  ```xml
  <!ELEMENT invoice (client, item+)>
  <!ATTLIST invoice num NMTOKEN #REQUIRED>
  <!ELEMENT client (name, email?)>
  <!ATTLIST client num NMTOKEN #REQUIRED>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT item (#PCDATA)>
  <!ATTLIST item
    price NMTOKEN #REQUIRED
    unit (FIM | EUR) "EUR" >
  ```

## Element type declarations

- **The general form is**
  ```xml
  <!ELEMENT elementType Name (E) >
  ```
  where `$E$` is a *content model*

- **Regular expression of element names**
  ```xml
  E | F : alternation
  E* : zero or more
  E+ : one or more
  (E) : grouping
  ```

## Attribute-List Declarations

- **Can declare attributes for elements:**
  - Name, data type and possible default value:
    ```xml
    <!ATTLIST FIG
      id ID #IMPLIED
      descr CDATA #REQUIRED
      class (a | b | c) "a">
    ```

## Mixed, Empty and Arbitrary Content

- **Mixed content:**
  ```xml
  <!ELEMENT P (#PCDATA | I | IMG)*>
  ```
  - may contain text (#PCDATA) and elements

- **Empty content:**
  ```xml
  <!ELEMENT IMG EMPTY>
  ```

- **Arbitrary content:**
  ```xml
  <!ELEMENT HTML ANY>
  (= <!ELEMENT HTML (#PCDATA | choice-of-all-declared-element-types) *>)
  ```

## Entities (1)

- **Named storage units or XML fragments (~ macros in some languages)**
  - **character entities:**
    ```xml
    &lt;: &amp;lt; &amp;#60; and &amp;#x3C; all expand to &lt;
    ```
    (treated as data, not as start of markup)
  - **other predefined entities:**
    ```xml
    &amp;gt; &amp;#62; and &amp;apos; &amp;#39; all expand to &gt; (treated as data)
    ```
    ```xml
    &amp;quot; &amp;#34; and &amp;#39; all expand to " (treated as data)
    ```
  - **general entities** are shorthand notations:
    ```xml
    <!ENTITY UKU "University of Kuopio"/>
    ```

## Entities (2)

- **physical storage units comprising a document**
  - **parsed entities**
    ```xml
    <!DOCTYPE doc [
      <!ENTITY chap1 SYSTEM "http://myweb/ch1">
    ]>
    ```
    - document entity is the starting point of processing
  - **entities and elements must nest properly**

## Unparsed Entities

- **For connecting external binary objects to XML documents:**
  - **Declaring entities**
    ```xml
    <!NOTATION TIFF "bin/gimp">
    <!ENTITY fig123 SYSTEM "figs/f123.tif" NDATA TIFF>
    ```
  - **Usage:**
    ```xml
    <IMG file="fig123"/>
    ```
    - application receives information about the notation
### Parameter Entities

- Way to parameterize and modularize DTDs

```
ENTITY % tableDTD SYSTEM "dtds/tab.dtd"
%tableDTD: <!-- include this
sub-dtd -->
ENTITY % stAttr "ready (yes | no) 'no'">
<ATTLIST CHAP %stAttr; >
```

### Speculations about XML Parsing

- Parsing involves two things:
  1. Pulling the entities together, and checking the syntactic correctness of the input
  2. Building a parse tree for the input (a’la DOM), or otherwise informing the application about document content and structure (e.g. a’la SAX)

- Task 2 is simple (← simplicity of XML markup; see next)

- Slightly more difficult to implement are
  - checking the well-formedness
  - checking the validity wrt the DTD (or a Schema)

### Building an XML Parse Tree

![XML Parse Tree Diagram]

- Check that the root is labelled by start symbol S of G
- Check that BCD matches the content model for A
- For each element node n of the tree, check that:
  - the attributes of n match attributes declared for the type of n
  - the contents of n matches the content model for the type of n.
  That is, if n is of type A and its children of type B₁,..., Bₙ, check that
```
word B₁... Bₙ ∈ L(E)  (1)
```

### Sketching the validation of XML

- How to check this?
  - Check that the root is labelled by start symbol S of G
  - For each element node n of the tree, check that:
```
word B₁... Bₙ ∈ L(E)  (1)
```

### Sketching the validation... (2)

- How to check condition (1; matching of children with a content model)?
  - by an automaton built from content model E
- Normally, validation proceeds in document order (= depth-first order of tree). For example, to validate the content of
```
<A><B>...</B><C>...</C><D/></A>
```
the content of B is checked before continuing to verify that BCD matches the content model for A

### 2.4 XML Namespaces

- Document often comprise parts processed by different applications (and/or defined in different grammars)
- For example, in XSLT scripts:
```
<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates />
  </h1>
</xsl:template>
```
- How to manage multiple sets of names?

### XML Namespaces (2/5)

- Solution: XML Namespaces, W3C Rec. 14/1/1999 for separating possibly overlapping “vocabularies” (sets of element type and attribute names) within a single document
- by introducing (arbitrary) local name prefixes, and binding them to (fixed) globally unique URIs
  - For example, the local prefix “xsl:” conventionally used in XSLT scripts

### XML Namespaces briefly (3/5)

- Namespace identified by a URI (through the associated local prefix)
  - e.g. http://www.w3.org/1999/XSL/Transform for XSLT
  - conventional but not required to use URLs
  - the identifying URI has to be unique, but it does not have to be an existing address
- Association inherited to sub-elements
  - see the next example (of an XSLT script)
XML Namespaces (4/5)

```xml
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/xhtml1/strict">
  <!-- XHTML is the 'default namespace' -->
  <xsl:template match="doc/title">
    <H1>
      <xsl:apply-templates />
    </H1>
  </xsl:template>
</xsl:stylesheet>
```

XML Namespaces briefly (5/5)

- Mechanism built on top of basic XML
  - overloads attribute syntax (xmlns:) to introduce namespaces
  - does not affect validation
    - namespace attributes have to be declared for DTD-validity
    - all element type names have to be declared (with their initial prefixes!)
  - Other schema languages (XML Schema, Relax NG) better for validating documents with Namespaces