2 Document Instances and Grammars

Fundamentals of hierarchical document structures, or
Computer Scientist's view of XML

2.1 XML and XML documents
2.2 Basics of document grammars
2.3 Basics of XML DTDs
2.4 XML Namespaces
2.5 XML Schemas

What is XML?

■ Extensible Markup Language is not a markup language!
  – does not fix a tag set nor its semantics
  (like markup languages like HTML do)
■ XML documents have no inherent (processing or presentation) semantics
  – How to specify and implement those semantics is the topic of this course!

How does it look?

<?xml version='1.0' encoding="iso-8859-1" ?>
<invoice num="1234">
  <client clNum="00-01">
    <name>Pekka Kilpeläinen</name>
    <email>kilpelai@cs.uku.fi</email>
  </client>
  <item price="60" unit="EUR">
    XML Handkook
  </item>
  <item price="350" unit="FIM">
    XSLT Programmer's Ref
  </item>
</invoice>

Essential Features of XML

■ Overview of XML essentials
  – many details skipped
    » some to be discussed in exercises or with other topics when the need arises
  – Learn to consult original sources (specifications, documentation etc) for details!
    » The XML spec is concise and readable
XML document characters

- XML documents are made of ISO-10646 (32-bit) characters; in practice of their 16-bit Unicode subset (used, e.g., in Java)
  - Unicode 2.0 defines almost 39000 distinct coded characters
- Characters have three different aspects:
  - their identification as numeric code points
  - their representation by bytes
  - their 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
  - for Latin-1 (8-bit Western European ASCII) use encoding="iso-8859-1"
- A font (collection of character images called glyphs) determines the visual presentation of characters

Documents are stored/transmitted as a sequence of bytes (of 8 bits). An encoding determines how characters are represented by bytes.

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
  - for Latin-1 (8-bit Western European ASCII) use encoding="iso-8859-1"
- 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 attr1="value1" … attrn="valuen">…</A>
    * names must be unique: attrk=attrh 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 (e.g.) <br/>
  - unique root element — document a single tree
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

- **Also:**
  - <!-- comments outside other markup -->
  - <!-- note this would be passed to the application as a processing instruction named 'note' -->

CDATA Sections

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

Two levels of correctness

- **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)

An XML Processor (Parser)

- Reads XML documents and reports their contents to an application
  - relieves the application from details of markup
  - XML Recommendation specifies partly the behaviour of XML processors,
    - recognition of characters as markup or data, what information to pass to applications,
    - how to check the correctness of documents:
      - validation based on comparing document against its grammar

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,

- **Many uses:**
  - text searching patterns (e.g., emacs, ed, awk, grep, Perl)
  - as a part of grammatical formalisms (for programming languages, XML, in XML DTDs and schemas, ...)

- **Relevance for document structures:** what kind of structural content is allowed within various 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 $a \in \Sigma$,
    - $(R \ | \ S)$ choice,
    - $(R \ S)$ concatenation, or
    - $(R)^*$ 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:
    - \((E) = E\)
  - binary operations are associative:
    - \((A \cdot (B \cdot C)) = ((A \cdot B) \cdot C) = (A \cdot B) \cdot C\)
  - operations have priorities:
    - iteration first, concatenation next, choice last
    - for example, \((A \cdot (B \cdot C)) \equiv A \cdot (B \cdot C)\)

Regular Expressions: Examples

- \(L(A \mid B \cdot (C \cdot D^*)) = \?)
  - \(L(A) \cup L(B \cdot (C \cdot D^*))\)
  - \((A) \cup L(B) \cdot L(C \cdot D^*)\)
  - \((A) \cup (B) \cdot L(C \cdot D^*)^* = (A) \cup (B \cdot (C \cdot D)^*)^*\)
  - \((A) \cup (B) \cdot \{A, CD, CD\cdot CD, CD\cdot CD\cdot CD, \ldots\}\)
  - \((A, B, B\cdot CD, B\cdot CD\cdot CD, B\cdot CD\cdot CD\cdot CD, \ldots\)\)

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 \((\text{nullikko}) (V, \Sigma, P, S)\)
  - \(V\) is the \textbf{alphabet} of the grammar \(G\)
  - \(\Sigma \subseteq V\) is a set of \textbf{terminal symbols} (päällesymbolit)
  - \(N = \Sigma \setminus \Sigma\) is a set of \textbf{nonterminal symbols} (väikesymbolit)
  - \(P\) set of \textbf{productions}
  - \(S \subseteq V\) the \textbf{start symbol} (lähtösymboli)

Productions and Derivations

- Productions: \(A \rightarrow \alpha, \text{ where } A \in N, \alpha \in V^*\)
  - E.g.: \(A \rightarrow a\beta_a\) (production \(P\))
- Let \(\gamma, \delta \in V^*\). String \(\gamma\) derives \(\delta\) directly, \(\gamma \Rightarrow \delta\), if
  - \(\gamma = \alpha\beta, \delta = \alpha\beta\) for some \(\alpha, \beta \in V^*\),
  - and \(A \rightarrow \alpha \in P\)
  - E.g. \(AA \Rightarrow AA\alpha\alpha\) (assuming production 1 above)

NB: CFGs are often given simply by listing the productions \(P\)
Language Generated by a CFG

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

Syntax Trees

- Also called parse trees (jäsennyspuu) or derivation trees
- Ordered trees
  - 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 with label $A$ can have children labelled by $X_1, \ldots, X_k$ only if $A \Rightarrow X_1, \ldots, X_k$

Syntax Trees: Example

CFG for simplified arithmetic expressions:

$V = \{E, +, *, I\}$; $S = \{+\}$; $N = \{E\}$; $S = E$

$P = \{E \Rightarrow E + E, E \Rightarrow E \times E, E \Rightarrow I, E \Rightarrow (E)\}$

Syntax tree for $2 \times (3 + 4)$?

CFGs for Document Structures

Nonterminals represent document elements
- E.g. model for items (Ref) of a bibliography list:
  - Right-hand-side of a CFG production is a fixed string of grammar symbols
  - Repetition simulated using recursion

Example: List of Three Authors

AuthorList

Aho Hopcroft Ullman
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 $\rightarrow$ Author* Title PubData
- Let $\gamma, \delta \in V^*$. String $\gamma$ derives $\delta$ directly, $\gamma \Rightarrow^* \delta$, if
  - $\gamma = \alpha \delta \beta$ for some $\alpha, \beta \in V^*$, and $A \rightarrow E \in P$ such that $\delta \in L(E)$
  - E.g., Ref $\Rightarrow^*$ Author Author Author Title PubData

Language Generated by an ECFG

- $L(G)$ defined similarly to CFGs:
  - $\gamma$ derives $\delta$, $\gamma \Rightarrow^* \delta$, if
    $\gamma \Rightarrow_0 \alpha_1 \Rightarrow \ldots \Rightarrow_0 \alpha_n \Rightarrow \delta$ (for $n \geq 0$)
  - $L(G) = \{ w \in V^* \mid \exists v \in V^* \text{ such that } w \}$
- Theorem: Extended and ordinary CFGs allow to generate the same (string) languages.

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

Syntax Trees of an ECFG

- Similar to parse trees of an ordinary CFG, except that ..
  - node with label A can have children labelled $X_1, \ldots, X_k$ when $A \rightarrow E \in P$ such that $X_1, \ldots, X_k \in L(E)$
  - an internal node may have arbitrarily many children (e.g., Authors below a Ref node)

Example: Three Authors of a Ref

- Ref $\rightarrow$ Author* Title PubData $\in P$
- Author Author Author Title PubData $\in L$(Author* Title PubData)

Terminal Symbols in Practise

- In (extended) CFGs:
  - Leaves of parse trees are labelled by single terminal symbols ($\epsilon \Sigma$)
- Too granular for practise; instead terminal symbols which stand for all values of a type
  - XML DTDs: #PCDATA for variable length string content
  - In XSDL (XML Schema Definition Language):
    - string, byte, integer, boolean, date, ...
  - Explicit string literals rare in document grammars
2. Basics of XML DTDs

A Document Type Declaration provides a grammar (document type definition, DTD) for a class of documents [Defined in XML Rec]

Syntax (in the prolog of a document instance):

```xml
<!DOCTYPE rootElement SYSTEM "ex.dtd">
<!-- "external subset" in file ex.dtd -->
<!-- "internal subset" may come here -->
>
```

- DTD is the union of the external and internal subset; internal subset 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 to the application

### How do Declarations Look Like?

```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)>
<!ATTLIST item price NMTOKEN #REQUIRED
unit (FIM | EUR) "EUR" >
```

### Element type declarations

- The general form is
  ```xml
  <!ELEMENT elementType [(E)]]
  ```
  where E is a content model
- regular expression of element names
- Content model operators:
  - E | F : alternation
  - E, F: concatenation
  - E? : optional
  - 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
- Example:
  ```xml
  <!ATTLIST FIG
  id ID #IMPLIED
desc CDATA #IMPLIED
class (a | b | c) "a">```
- Semantics mainly up to the application
  - processor checks that ID attributes are unique and that targets of IDREF attributes exist

### Mixed, Empty and Arbitrary Content

- Mixed content:
  ```xml
  <!ELEMENT TEXT (TEXT | I | IMG)*)
  ```
  - may contain text (#PCDATA) and elements
- Empty content:
  ```xml
  <!ELEMENT IMG EMPTY>
  ```
- Arbitrary content:
  ```xml
  <!ELEMENT HTML ANY>
  ```
  (= !ELEMENT HTML (TEXT | I | IMG)*)

### Mixed, Empty and Arbitrary Content

- Can declare attributes for elements:
  - Name, data type and possible default value
- Example:
  ```xml
  <!ATTLIST FIG
  id ID #IMPLIED
desc CDATA #IMPLIED
class (a | b | c) "a">```
- Semantics mainly up to the application
  - processor checks that ID attributes are unique and that targets of IDREF attributes exist
Entities (1)

- Physical storage units or named fragments of XML documents
- Multiple uses:
  - character entities:
    - `&gt;`, `&lt;`, `&apos;`, `&quot;`; all expand to `<`
    (treated as data, not as start-of-markup)
  - other predefined entities:
    - `&amp;`, `&apos;`, `&quot;`, `&lt;`
    expand to `&`, `'`, `'` and "`
  - general entities are shorthand notations:
    - `&lt;ENTITY UKU "University of Kuopio"></ENTITY>`

Entities (2)

- physical storage units comprising a document
  - parsed entities
    - `&lt;ENTITY chap1 SYSTEM "http://myweb/ch1"></ENTITY>`
  - document entity is the starting point of processing
    - entities and elements must nest properly:
      - `&lt;DOCTYPE doc [ ... as above ... ] &gt; &lt;/sec&gt; &lt;/doc&gt; &lt;/sec&gt; &lt;/sec&gt; &lt;/sec&gt; &lt;/sec&gt;`
  - `&lt;sec num="1"> ... &lt;/sec&gt; &lt;sec num="2"> ... &lt;/sec&gt; &lt;/sec&gt; &lt;/sec&gt; &lt;/sec&gt;`

Unparsed Entities

- For connecting external binary objects to XML documents; (XML processor handles only text)
- Declarations:
  - `<&NOTATION TIFF "bin/xv"></&NOTATION>`
  - `<ENTITY fig123 SYSTEM "figs/f123.tif" NDATA TIFF>`
  - `<&ATTLIST IMG file ENTITY #REQUIRED>`
- Usage: `<IMG file="fig123"/>`
  - application receives information about the notation

Parameter entities

- Way to parameterize and modularize DTDs
  - `<ENTITY % table-dtd SYSTEM "dtds/tab.dtd"> &table-dtd &lt;!-- include external sub-dtd --&gt;```
  - `<ENTITY % stAttr "status (draft | ready) --&gt;```
  - `<&ATTLIST CHAP %stAttr; >`
  - `<&ATTLIST SECT %stAttr; >`
  - (The latter, parameter entities as a part of a markup declaration, is allowed only in the external, and not in the internal DTD subset)

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
<S>Hello</S>
```

8
Sketching the validation of XML

- Treat the document as a tree of
- Document is valid w.r.t. a grammar (DTD/Schema) G if d is a syntax tree over G
- 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
    - 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)
  for some production A → E of G.

2.4 XML Namespaces

- Sometimes we need to combine document parts processed by different applications (and/or defined in different grammars) in a single document instance
  - 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
  - separation by using a local name prefix, which is bound to a globally unique URI
  - For example, the local prefix "xsl:" conventionally used in XSLT scripts

XML namespces briefly (3/5)

- Namespace identified by a URI (through the associated local prefix)
  - e.g. http://www.w3.org/1999/XSLT/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 Namespaces briefly (5/5)

- Namespace mechanism is built on top of basic XML features
  - uses attribute syntax (`xmlns`) to introduce namespaces
  - does not affect validation
    - namespace attributes have to be declared for DTD-validation
    - all element type names have to be declared in the DTD (with their initial prefixes)
- > Namespaces and DTD-validation do not mix well