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
<?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 Handbook</item>
  <item prices="350" unit="FIM">
    XML Programmer’s Ref</item>
</invoice>
```

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, a reformation of HTML using XML
- Often “XML” = XML + XML technology
  - that is, processing models and languages we’re studying

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

External Aspects of 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

XML document 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 Encodings of Structure

- 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 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" ... attrk="valuek"/>
  – names must be unique: attrif=attri, when k ≠ h
  – text nodes written as their string value

XML Encoding of Structure: Example

```
<W>S</W>Hello</W></E><W>world!</W></S>
```

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
    `<div class="preface" date='990126'> ... </div>`
  – **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

```
<![CDATA[
  Here we can easily include markup characters and, for example, code fragments:
  <example if (Count < 5 && Count > 0)
  ]]>>
```

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)

XML docs and valid XML docs

- XML documents = well-formed XML documents

```
An XML Processor (Parser)

- Reads XML documents and reports their contents to an application
  - relieves the application from details of markup
  - XML Recommendation specifies, partially, 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

Next: Basics of document grammars

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 Σ is either
  - ∅ an empty set,
  - λ lambda (sometimes epsilon ∈),
  - a any alphabet symbol a ∈ Σ,
  - (R | S) choice,
  - (R S) concatenation, or
  - (R)* Kleene closure or iteration,
- N.B: different syntaxes exist but the idea is same

Regular Expressions: Semantics

- Regular expression E denotes a language (set of strings) L(E), defined inductively as follows:
  - \( L(∅) = \{ \} \) (empty set)
  - \( L(λ) = \{ λ \} \) (singleton set of empty string λ = “”)
  - \( L(a) = \{ a \} \), \( a ∈ Σ \) (singleton set of word “a”)
  - \( L((R | S)) = L(R) \cup L(S) \) \( (w | v ∈ L(R) or w ∈ L(S)) \)
  - \( L((R S)) = L(R) L(S) \) \( \{ xy | x ∈ L(R) and y ∈ L(S) \} \)
  - \( L(R^*) = L(R)^* \) \( \{ x_1 \ldots x_n | x_k ∈ L(R), k = 1, \ldots, n, n ≥ 0 \} \)

Regular Expressions: Examples

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

Regular Expressions: Grouping

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

Context-Free Grammars (CFGs)

- Used widely for syntax specification (programming languages, XML, …) and to parser/compiler generation (e.g., YACC/GNU Bison)
- CFG G formally a 4-tuple (nelikko) \( (V, Σ, P, S) \)
  - \( V \) is the alphabet of the grammar \( G \)
  - \( Σ ⊂ V \) is a set of terminal symbols (päätetsymbolit);
  - \( N = V - Σ \) is a set of nonterminal symbols (välikesymbolit)
  - \( P \) set of productions
  - \( S ∈ V \) the start symbol (lähtössymboli)
Productions and Derivations

- Productions: $A \rightarrow \alpha$, where $A \in N$, $\alpha \in V^*$
  - E.g.: $A \rightarrow ab\alpha$ (production 1)
- Let $\gamma, \delta \in V^*$. String $\gamma$ derives $\delta$ directly, $\gamma \Rightarrow \delta$ if
  - $\gamma = \alpha\beta$, $\delta = \alpha\beta\beta$ for some $\alpha, \beta \in V^*$,
  - and $A \rightarrow \alpha \in P$
- E.g.: $A \Rightarrow A\beta\alpha$ (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 (johtopuu)
- 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 $A \rightarrow X_1 \ldots X_k \in P$

CFGs for Document Structures

- Nonterminals represent document elements
  - E.g. model for items $(Ref)$ of a bibliography list: $Ref \rightarrow \text{AuthorList Title PublData}$
  - $\text{AuthorList} \rightarrow \text{Author AuthorList}$
  - $\text{AuthorList} \rightarrow \lambda$
- 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

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

Problems
Extended CFGs (ECFGs)

- Like CFGs, but right-hand-sides of productions are regular expressions over V
  - E.g.: \( \text{Ref} \rightarrow \text{Author}^{*} \text{Title} \text{PublData} \)
- Let \( \gamma, \delta \in V^* \). String \( \gamma \) derives \( \delta \) directly, \( \gamma \Rightarrow \delta \), if
  - \( \gamma = \alpha A \beta \), \( \delta = \alpha \alpha \beta \) for some \( \alpha, \beta \in V^* \), and \( A \rightarrow e \in P \) such that \( e \in L(E) \)
- E.g.: \( \text{Ref} \rightarrow \text{Author}^{*} \text{Author}^{*} \text{Title} \text{PublData} \)

Language Generated by an ECFG

- \( L(G) \) defined similarly to CFGs:
  - \( \gamma \) derives \( \delta, \gamma \Rightarrow \delta, \) if
    - \( \gamma \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow \delta \) (for \( n \geq 0 \))
    - \( L(G) = \{ w \in \Sigma^* | S \Rightarrow^* w \} \)
- 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 \( X_1, \ldots, X_n \) when \( A \rightarrow E \in P \) such that \( X_1, \ldots, X_n \in L(E) \)
  - an internal node may have arbitrarily many children (e.g., Authors below a Ref node)

Example: Three Authors of a Ref

![Example](image)

Terminal Symbols in Practise

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

2. 3 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 rootElemType 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+)>
<!ELEMENT invoice num NMTOKEN #REQUIRED>
<!ELEMENT client (name, email?)>
<!ATTLIST invoice num NMTOKEN #REQUIRED>
<!ATTLIST client name (#PCDATA)>
<!ATTLIST invoice email (#PCDATA)>
<!ELEMENT item (#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
- No priorities: either (A,B)|C or A,(B|C), but not A,B|C

Attribute-List Declarations

- Can declare attributes for elements:
  - Name, data type and possible default value
- Example:
  ```xml
  <!ATTLIST FIG
    id ID #IMPLIED
    descr CDATA #REQUIRED
  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 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)

- Physical storage units or named fragments of XML documents
- Multiple uses:
  - character entities:
    ```xml
    &amp; &gt; &apos; &quot;
    ```
    - other predefined entities:
      ```xml
      &amp;lt; &amp;apos; &amp;quot;
      ```
  - general entities are shorthand notations:
    ```xml
    <!ENTITY UKU "University of Kuopio">
    ```

Entities (2)

- physical storage units comprising a document
  - parsed entities
    ```xml
    <!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; (XML processor handles only text)
- Declarations:
  ```xml
  <!NOTATION TIFF "bin/gimp">
  ```
  - NDATA TIFF
  ```xml
  <!ATTLIST IMG
    file ENTITY #REQUIRED>
  ```
- Usage:
  ```xml
  <IMG file="fig123"/>
  ```
  - application receives information about the notation

Parameter entities

- Way to parameterize and modularize DTDs
  ```xml
  <!ENTITY % table-chap SYSTEM "dtds/tab.dtd">
  ```
  %table-chap;
  - include external sub-dtd -->
  ```xml
  <!ENTITY % stAttr "status (draft | ready)"

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 (a’la 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)
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/></A>
the content of B is checked before continuing to verify that BCD matches the content model for A