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

SDPL 2007 2: Document Instances and Grammars

2.1 XML and XML documents

  - not an official standard, but a stable industry standard
    > editorial revisions, not new versions of XML 1.0
  - what is said later about valid XML documents applies to SGML documents, too

SDPL 2007 2: Document Instances and Grammars

2 Document Instances and Grammars

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

SDPL 2007 2: Document Instances and Grammars

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 reformulation of HTML using XML
- Often ‘XML = XML + XML technology’
  - that is, processing models and languages we’re studying (and many others ...)

SDPL 2007 2: Document Instances and Grammars

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 specification is easy to browse

SDPL 2007 2: Document Instances and Grammars

Semantics of XML Markup

- Meaning of this XML fragment?
  - Customer
  <invoice>
    <item price="60' unit="EUR">
      XML Handbook</item>
  <item price="350' unit="FIM">
    XHTML Programmer’s Ref</item>
  </invoice>

SDPL 2007 2: Document Instances and Grammars

How does it look?

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)
- Three aspects or characters:
  - identification as numeric code points
  - representation by bytes
  - visual presentation

SDPL 2007 2: Document Instances and Grammars
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 (ISO 8859-1) is the XML default encoding
  - encoding="iso8859-1" --> 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:
    \(<A \text{ attr}="\text{value}\" ... \text{ attr}_r="\text{value}\"/>\)
  - names must be unique: \(\text{attr}_r \neq \text{attr}_s\) when \(k \neq h\)
  - text nodes written as their string value

XML: Logical Document Structure

- Elements
  - indicated by matching (case-sensitive!) tags
  - can contain text and/or subelements
  - can be empty:
    \(<\text{elem-type}/\text{elem-type}>\) or \(<\text{elem-type/>\text{elem-type}>\) (e.g. \(<br/>\) in XHTML)
  - 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
  - \(<\text{div class="preface" date='990126'/>\) ...
- Also:
  - \(<!---- \text{comments outside other markup} --->\)
  - \(<\text{note this would be passed to the application as a processing instruction named 'note'}/>\)

CDATA Sections

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

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

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

![Diagram showing XML documents and valid 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, 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, ...
- 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 \( a \in \Sigma \),
  - \( (R | S) \) 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:
    - \( L(\emptyset) = \emptyset \)
    - 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, \( (A | (B | C)^*) = A | B | C^* \)

Regular Expressions: Examples

- \( L(A | B | (C D)^*) = ? \)
  - \( L(A) \cup L(B | (C D)^*) \)
  - \( \{A\} \cup L(B) L((C D)^*) \)
  - \( \{A\} \cup \{B\} L(C D)^* = \{A\} \cup \{B\} \{CD\}^* \)
  - \( \{A\} \cup \{B\} \{\lambda, CD, CDCCD, CDCCDCD, \ldots\} \)
  - \( \{A, B, BCD, BCDCD, BCDCDCD, \ldots\} \)

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* date sect sect*
- Commonly used abbreviations:
  - \( E? = (E | \lambda) \)
  - \( E^* = E | E^+ \)
  - The above more compactly:
    - title auth* date? sect+
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 \((V, \Sigma, P, S)\)
  - \(V\) is the alphabet of the grammar \(G\)
  - \(\Sigma \subset V\) is a set of terminal symbols (päästysymbolit)
  - \(N = V \setminus \Sigma\) is a set of nonterminal symbols (valikkesymbolit)
  - \(P\) set of productions
  - \(S \in V\) the start symbol (alusta)

Productions and Derivations

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

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 \(G\):
  - \(L(G) = \{ w \in \Sigma^* | S \Rightarrow^* w \}\)
- 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:
  \[V = \{E, +, \ast, I\}; \Sigma = \{+, \ast, I\}; N = \{E\}; S = E\]
  \((I\) stands for an arbitrary integer)
  \[P = \{ E \rightarrow E+E,\ E \rightarrow E\ast E,\ E \rightarrow I,\ E \rightarrow (E) \}\]
- Syntax tree for \(2*(3+4)\)?

Syntax Trees: Example

- Example: List of Three Authors

CFGs for Document Structures

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

Example: List of Three Authors

- AuthorList
  - Author
    - Ref
  - AuthorList
    - Author
  - AuthorList
    - Author
  - AuthorList
    - Author
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 = α1β, δ = α2β for some α, β ∈ V*, and A → E ∈ P such that ω ∈ L(E)
  - E.g. Ref -> Author 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 (for n ≥ 0)
    L(G) = {w ∈ Σ* | S =>* w }
- Theorem: Extended and ordinary CFGs allow to generate the same (string) languages.

Syntax Trees of an ECFG

- Similar to parse trees of an ordinary CFG, except that . . .
  - node with label A can have children labelled X1, . . . , Xn when A → E ∈ P such that
    X1 . . . Xn ∈ L(E)
  - an internal node may have arbitrarily many children (e.g., Authors below a Ref node)

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:
      - string, byte, integer, boolean, date, . . .
      - Explicit string literals rare in document grammars

Terminal Symbols in Practise

- E.g.:
  - Author Author Author Title PublData ∈ L(Author* Title PublData)

DTD/ECFG Correspondence

- 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):
  - <!DOCTYPE rootElementType SYSTEM "ex.dtd"
  - "external subset" in file ex.dtd -->
  - "internal subset" may come here -->
- 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 to the application

Element type declarations

- The general form is
  `<!ELEMENT elementTypeName (E)>`
  where E is a content model
- regular expression of element names
- Content model operators:
  E | F : alternation
  E* : zero or more
  E+ : one or more
  (E) : grouping
- No priorities: either A,B|C or A,B(C), but no A,B|C

Mixed, Empty and Arbitrary Content

- Mixed content:
  `<!ELEMENT P (#PCDATA | I | IMG)>`
  - may contain text (#PCDATA) and elements
- Empty content:
  `<!ELEMENT IMG EMPTY>`
- Arbitrary content:
  `<!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:
    » &lt; &amp;gt;; and &amp;#x3C; all expand to '<'
      (treated as data, not as start-of-markup)
    » other predefined entities:
      &amp;lt; &amp;gt; &apos; &quot; for & &
  - general entities are shorthand notations:
    `<!ENTITY URO "University of Kuopio">`

Entities (2)

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

How do Declarations Look Like?

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

Unparsed Entities

- For connecting external binary objects to XML documents; (XML processor handles only text)
  - Declarations:
    `<!NOTATION TIFF "bin/gimp">`
    `<!ENTITY fig123 SYSTEM "figs/f123.tif">`
    `<!ATTLIST IMG
      file ENTITY #REQUIRED>`
  - Usage: `<IMG file="fig123"/>`
    - parser provides notation and address to the application
An Alternative Way

- I have rarely used unparsed entities or notations
- Easier "HTML-style" linking:

  ```xml
  <IMG type="TIFF" file="figs/fig123"/>
  ```

  – the link indicates the format and the address directly

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

- How to check this?
  - Normally, validation proceeds in document order (= depth-first order of tree). For example, to validate the content of

  ```xml
  <A><B>...
  ```

  - the content of B is checked before continuing to verify that BCD matches the content model for A

Sketching the validation of XML

- Treat the document as a tree d
  - Document is valid w.r.t. a grammar (DTD/Schema) G iff 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 B<sub>1</sub>, ..., B<sub>n</sub>, check that

  ```latex
  B_1 \ldots B_n \in L(E)
  ```

  for some production A → E of G.

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

  ```xml
  <A><B>
  ```

  - the content of B is checked before continuing to verify that BCD matches the content model for A

XML Namespaces

- 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

2.4 XML Namespaces

- Document often comprise parts processed by different applications (and/or defined in different grammars)
  - for example, 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 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

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>
```