2 Document Instances and Grammars

2.1 XML and XML documents

  - not an official standard, but a stable industry standard
  - an editorial revision, not a new version of XML 1.0
- a simplified subset of SGML, Standard
  - what is said later about valid XML documents applies to SGML documents, too

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 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 reformulation of HTML using XML

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 price="350" unit="FIM">
    XSLT Programmer’s Ref
  </item>
</invoice>
```

Essential Features of XML

- We’ll have just an overview of the essentials of XML
  - many details skipped
  - some to be discussed in exercises or with other topics when the need arises
  - learn to consult the 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., 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

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 (= namestring-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_1="value_1" … attr_n="value_n">`
  - text nodes written as their string value

XML Encoding of Structure: Example

- `<S>` `<W>` Hello `<W>` world! `<W>` `</E>` `<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'>` ...
- 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
  ```xml
  <![CDATA[
  Here we can easily include markup characters and, for example, code fragments:
  <example>if (Count < 5 && Count > 0)
  <![CDATA[
  ]]>]
  ```

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 Rec 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 over an alphabet Σ is either
  - ∅ an empty set,
  - λ lambda (sometimes epsilon ε),
  - a any alphabet symbol a ∈ Σ,
  - (R | S) choice; sometimes (R ∪ S),
  - (RS) 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 | (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

N.B: different syntaxes exist but the idea is same
Regular Expressions: Semantics

- A regular expression \( E \) denotes a language \( L(E) \), defined inductively as follows:

  - \( L(\emptyset) = \{\} \) (empty set)
  - \( L(\lambda) = \{\lambda\} \) (singleton set of empty string \( \lambda = '' \))
  - \( L(a) = \{a\} \), \( a \in \Sigma \), singleton set of word "a"
  - \( L(R | S) = L(R) \cup L(S) \)
  - \( L(R S) = \{xy \mid x \in L(R) \land y \in L(S)\} \)
  - \( L(R^*) = L(R)^* = \{x_1 \ldots x_n \mid x_k \in L(R), k = 1, \ldots, n; n \geq 0\} \)

Regular Expressions: Examples

- Simplified top-level structure of a document:
  - \( \Sigma = \{\text{title, auth, date, sect}\} \)
  - \( \text{title} \) followed by an optional list of authors, by an optional date, by one or more sections:
    - \( \text{title auth* date? sect+} \)

Commonly used abbreviations:

- \( E^? = (E | \lambda) \)
- \( E^+ = EE^* \)

The above more compactly:

- \( \text{title auth* date? sect+} \)

Context-Free Grammars (CFGs)

- Used widely to syntax specification (programming languages, XML, ...) and to parser/compiler generation (e.g., YACC/GNU Bison)

- A CFG formally a 4-tuple \((V, \Sigma, P, S)\):
  - \( V \) is the alphabet of the grammar \( G \)
  - \( \Sigma \subseteq V \) is a set of terminal symbols; \( N = V - \Sigma \) is a set of nonterminal symbols
  - \( P \) set of productions
  - \( S \in V \) the start symbol

Productions and Derivations

- Production: \( A \rightarrow \alpha \), where \( A \in \Sigma, \alpha \in V^* \)
- \( \gamma \rightarrow \delta \) derivates \( \delta \) if \( \gamma = \alpha \mathbf{A} \beta \), \( \delta = \alpha \omega \beta \) for some \( \alpha, \beta \in V^* \), and \( \mathbf{A} \rightarrow \omega \in P \)

Language Generated by a CFG

- The language generated by a CFG \( G \):
  - \( L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\} \)

NB: CFGs are often given simply by listing the productions; the start symbol is then conventionally the left-hand-side of the first production.
Syntax Trees

- Also called parse trees or derivation trees
- Ordered trees: consist of nodes that may have child nodes which are ordered left-to-right
- Nodes 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 $A \rightarrow X_1, \ldots, X_k \in P$

Syntax Trees: Example

CFG for simplified arithmetic expressions:

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

$P = \{ E \rightarrow E+E, E \rightarrow E*E, E \rightarrow I, E \rightarrow (E) \}$

Syntax tree for $2*(3+4)$?

Syntax Trees: Example

CFGs for Document Structures

- Nonterminals represent document elements
  - E.g. model for items (Ref) of a bibliography list:
    $Ref \rightarrow AuthorListTitlePubData$
    $AuthorList \rightarrow AuthorAuthorList$
    $AuthorList \rightarrow \lambda$

Notice:
- Right-hand-side of a CFG production is a fixed string of grammar symbols
- Repetition simulated using recursion
  - E.g. $AuthorList$ above

Example: List of Three Authors

Problems

- "Auxiliary nonterminals" (like $AuthorList$) obscure the model
  - The last $Author$ several levels apart from its intuitive parent element $Ref$
  - Awkward 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.: $\text{Ref} \rightarrow \text{Author}^* \text{Title PublData}$

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

Language Generated by an ECFG

- $L(G)$ defined similarly to CFGs:
  - $\gamma$ derives $\delta$, $\gamma \Rightarrow^* \delta$, if
    - $\gamma \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n = \delta$ (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 $X_1, \ldots, X_k$ 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
Author Author Author Title PublData

Aho Hopcroft Ullman The Design and Analysis ...
```

Terminal Symbols in Practice

- In (extended) CFGs:
  - Leaves of parse trees are labelled by single terminal symbols ($\in \Sigma$)
  - 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 definition language:
      - w 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+)>
<!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 (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) : grouping

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
    &lt; &gt; &amp; #x2c; all expand to '<'
    ```
    (treated as data, not start-of-markup)
  - other predefined entities:
    ```xml
    &amp;amp; &amp;gt; &amp;apos; &amp;quot; expand to & , > , ' and "
    ```
  - **general entities** are shorthand notations:
    ```xml
    <!ENTITY UKU "University of Kuopio" >
    ```
Entities (2)

- Physical storage units comprising a document
  - Parsed entities
  - Document entity is the starting point of processing
  - Entities and elements must nest properly:

```xml
<!DOCTYPE doc [
  <!ENTITY chap1 SYSTEM "http://myweb/ch1">
]
<doc>
  &chap1;
</doc>
```

Unparsed Entities

- For connecting external binary objects to XML documents; (XML processor handles only text)
- Declarations:

```xml
<!NOTATION TIFF "bin/xv">
<!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

```xml
<!ENTITY % table-dtd SYSTEM "dtds/tab.dtd">
%table-dtd;  <!-- include external sub-dtd -->
<!ENTITY % stAttr "status (draft | ready) "draft">
<!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 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
<doc>
  <sec num="1">
  ...
  </sec>
  <sec num="2">
  ...
  </sec>
</doc>
```

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 match 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
    \[ \text{word } B₁ \ldots Bₙ \in L(E) \quad (1) \]
    for some production \( A \rightarrow E \) 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

  `<A><B>...<B><C>...<C/><D/></A>`

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

2.4 XML Namespaces

- Sometimes needed to combine document parts processed by different applications (and/or defined in different grammars) in a single document instance
  - for example, in XSLT scripts:

    ```xml
    <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 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 xsl: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)

- Namespaces 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-validity
    - all element type names have to be declared in the DTD (with their initial prefixes?)
  - Namespaces and DTD-validation do not mix well