2 Document Instances and Grammars

2.1 XML and XML documents

- 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

What is XML?

How does it look?

```
<?xml version='1.0' encoding='iso-8859-1' ?>
<invoice num='1234'>
  <client name='Fekta Kilpelainen'></client>
  <item price='660' unit='EUR'>
    <name>XML Handbook</name>
    <unitprice>350</unitprice>
    XML 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., 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 (8-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 attr="value" ... attr="value_n"/>
  - 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: <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 -->
  - <!-- 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)
  </example>
  ]]>
  ```

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 $\Sigma$ 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:
    - $(A | (B | C)) = (A | B) | C$
    - $(A | B) | C = A | (B | C)$
  - operations have priorities:
    - iteration first, concatenation next, choice last
    - for example:
      - $(A | (B | C))* 
      - $A | (B | C)^*$
Regular Expressions: Semantics

- Regular expression \( E \) denotes a language (set of strings) \( L(E) \), defined inductively as follows:
  - \( L(\emptyset) = \{\} \) (empty set)
  - \( L(\lambda) = \{\lambda\} \) (singleton set of empty string \( \lambda \) in \( \Sigma \))
  - \( L(a) = \{a\} \) (singleton set of word \( a \))
  - \( L(R \cup S) = L(R) \cup L(S) = \{w | w \in L(R) \text{ or } w \in L(S)\} \)
  - \( L(R \cdot S) = L(R) \cdot L(S) = \{xy | x \in L(R) \text{ and } y \in L(S)\} \)
  - \( L(R^*) = \{x_1 \cdots x_n | x_i \in L(R), k = 1, \ldots, n, n \geq 0\} \)

Regular Expressions: Examples

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

- Commonly used abbreviations:
  - \( E_n = (E \mid \lambda) \), \( E^* = E \cdot E^* \)
  - 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)

- 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:
    - \( N \subset V \) is a set of nonterminal symbols
  - \( P \) set of productions
  - \( S \in V \) the start symbol

Productions and Derivations

- Productions: \( A \rightarrow \alpha \), where \( A \in N, \alpha \in V^* \)
  - E.g.: \( A \rightarrow a\beta \) (production \( \beta \))

- Let \( y, \delta \in V^* \). String \( y \) derives \( \delta \) directly, \( y \Rightarrow \delta \), if
  - \( y = \alpha\beta, \delta = \alpha\delta \) for some \( \alpha, \beta \in V^* \),
  - and \( A \rightarrow \delta \in P \)
  - E.g.: \( AA \Rightarrow A\alpha A\beta \) (assuming production 1 above)

- NB: CFGs are often given simply by listing the productions \( P \)

Language Generated by a CFG

- \( y \) derives \( \delta \), \( y \Rightarrow^* \delta \), if there’s a sequence of \( 0 \text{ or more} \) direct derivations that transforms \( y \) 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
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 \( \mathcal{V} \):
  - 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:
  \[
  \mathcal{V} = \{ E, +, *, I \}; \quad \Sigma = \{ +, *, I \}; \quad N = \{ E \}; \quad S = E
  \]
  (\( I \) stands for an arbitrary integer)
  \[
  P = \{ E \rightarrow E + E, \\
  E \rightarrow E * E, \\
  E \rightarrow I, \\
  E \rightarrow (E) \}
  \]
- Syntax tree for \( 2^3(3+4) \)?

CFGs for Document Structures

- Nonterminals represent document elements
  - E.g., model for items (\( \text{Ref} \)) of a bibliography list:
    \[
    \text{Ref} \rightarrow \text{AuthorList Title PubData} \\
    \text{AuthorList} \rightarrow \text{Author AuthorList} \\
    \text{AuthorList} \rightarrow \lambda
    \]
- 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 \( \text{Author} \) several levels apart from its intuitive parent element \( \text{Ref} \)
  - Awkward 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 \( \mathcal{V} \)

- E.g. \( \text{Ref} \rightarrow \text{Author* Title PubData} \)

- Let \( \gamma, \delta \in \mathcal{V}^{*} \). String \( \gamma \) derives \( \delta \) directly, \( \gamma \Rightarrow \delta \), if
  - \( \gamma = \alpha \beta \), \( \delta = \alpha \omega \beta \) for some \( \alpha, \beta \in \mathcal{V}^{*} \),
  and \( \lambda \rightarrow E \in \mathcal{P} \) such that \( \omega \in \mathcal{L}(E) \)

- E.g. \( \text{Ref} \rightarrow \text{Author Author Author Title PubData} \in \mathcal{L}(\text{Author* Title PubData}) \)

Language Generated by an ECFG

- \( \mathcal{L}(E) \) defined similarly to CFGs:
  - \( \gamma \ ) derives \( \delta \) \( \Rightarrow \delta \), if
    - \( \gamma = \alpha_{1} \ldots \alpha_{n} \), \( \delta = \alpha_{1} \ldots \alpha_{n} \delta \) \( \text{for } n \geq 0 \)
    - \( \mathcal{L}(E) = \{ w \in \mathcal{V}^{*} \mid S \Rightarrow^{*} w \} \)

- 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 \( A \rightarrow E \in \mathcal{P} \) such that
    \( X_{1}, \ldots, X_{n} \in \mathcal{L}(E) \)

  - An internal node may have arbitrarily many children (e.g., Authors below a Ref node)

Example: Three Authors of a Ref

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 rootElType 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 ECFS
  - 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
  \[ \langle !ELEMENT \text{elementTypeName} \ (E) \rangle \]
  where \( E \) is a content model
- Content model operators:
  \( E | F: \text{alternation} \)
  \( E;: \text{optional} \)
  \( E^+: \text{one or more} \)
  \( (E): \text{grouping} \)

Attribute-List Declarations

- Can declare attributes for elements:
  - Name, data type and possible default value
- Example:
  \[ \langle !ATTLIST fig id ID \ #IMPLIED \ descr CDATA \ #REQUIRED \ class \ (a | b | c) \ "a" \rangle \]
- Semantics mainly up to the application
  - Processor checks that 10 attributes are unique and that targets of IDREF attributes exist

Mixed, Empty and Arbitrary Content

- Mixed content:
  \[ \langle !ELEMENT P \ [#PCDATA | I | IMG]* \rangle \]
  - may contain text (#PCDATA) and elements
- Empty content:
  \[ \langle !ELEMENT IMG \ EMPTY \rangle \]
- Arbitrary content:
  \[ \langle !ELEMENT HTML \ ANY \rangle \]
  \[ (= \langle !ELEMENT HTML \ [#PCDATA | choice-of-all-declared-element-types]* \rangle) \]

Entities (1)

- Physical storage units or named fragments of XML documents
- Multiple uses:
  - character entities:
    \[ &lt; \#60; \text{and} \ #63; \text{all expand to } 'c' \]
    (treated as data, not as start-of-markup)
  - other predefined entities:
    \[ &amp; \text{glt}; \#apos; \#quote; \]
    (expand to &, ' and "
  - general entities are shorthand notations:
    \[ \langle !ENTITY UUKU "University of Kuopio" \rangle \]
Entities (2)

- physical storage units comprising a document
  - parsed entities
    <ENTITY chap1 SYSTEM "http://myweb/blt">...
  - 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:
  <ENTITY fig123 SYSTEM "figs/fig123.tif" BASETYPE 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 "table/tab.dtd"/>
  <ENTITY & status.dtd SYSTEM "status/draft.dtd"/>
  <ENTITY & draft.dtd SYSTEM "status/ready.dtd"/>
  <ATTLIST CHAP hasAttr; > "draft";>
  <ATTLIST TEXT sect hasAttr; >
  (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 w.r.t. the DTD (or a Schema)


Building an XML Parse Tree

<node S><node W><node E><node W><node world!</node W></node S>

Sketching the validation of XML

- Treat the document as a tree d
- Document is valid w.r.t. a grammar (DTD/Scheme) 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.
    - If n is of type A and its children of type B₁,...,Bₙ check that
      - word B₁ · · · Bₙ ∈ L(F) (1)
      - 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 automation built from content model E
- Normally, validation proceeds in document order (= depth-first order of tree). For example, to validate the content of

  \[
  \langle A \rangle \langle B \rangle ... \langle /B \rangle \langle C \rangle ... \langle /C \rangle \langle D \rangle /A
  \]

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

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 &quot;xsl:” conventionally used in XSLT scripts

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
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml1/dtd/xhtml1-strict.dtd">
  <html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
    <head></head>
    <body>
      <h1>&lt;xi:template match="doc:title"&gt;&lt;h1&gt;\&lt;/h1&gt;&lt;/xi:template&gt;</h1>
    </body>
  </html>
  ```

- How to manage multiple sets of names?

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 URIs
  - 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
<xs:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:html="http://www.w3.org/TR/xhtml1/strict">
  <xs:template match="doc:title">
    <h1>xsl:apply-templates />
  </xs:template>
</xs: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