Syntax (XML & JSON)

Lecture:

- Aim of XML, JSON, etc:
  - Denormalize
  - Allow for heterogeneity
  - Data model: Tree
- JSON:
  - Datatypes:
    - String
    - Array
    - Object
    - Boolean
    - Null
    - Numbers
  - Adding simplicity

Introduction to XML (Chapter 1):

- Extensible Markup Language
- Cross-platform, long term data format, well-understandable, easy to parse, easily portable, very flexible!
- XML is a descendant of SGML, the Standard Generalized Markup Language
  - SGML was very successful, but too complicated, thus XML
  - After XML 1.0: Namespaces introduced —> allow have markup from several XML applications in one document without conflicts when names are occurring in both
  - Then: Extensible Stylesheet Language (XSL) for displaying XML in browser
- "metamarkup" language: No fixed set of tags, but rather allows developers to invent the elements that they need ("extensible": Can be extended to meet different needs)
- Individuals or organizations may agree to use only certain tags. These tag sets are called XML applications
- Schema:
  - Associate document with schema to check for validity.
  - Languages:
    - DTD: quite broad, limited Syntax
    - W3C XML Schema Language
    - Other languages: RELAX NG, Schematron, Hook, and Examplotron
  - All schema languages are purely declarative
- XML is not a programming language
  - No compiler that produces executable
  - Rather like a config file, as a format of instructions
  - But: One XML application, XSL Transformations is Turing complete by construction
- XML is not a database
XML is not a network transport protocol (unlike HTML):
- HTML only allows small set of tags, XML you can create any tags
- Structural and Semantic markup language, not a presentation language (tags only describe structure of the document, and semantic, e.g that the content of the tag is a person, but not how it should be displayed)

XML parser:
- Responsible for dividing the document into individual elements, attributes, and other pieces
- Reports error if detecting violation of wellformedness
- **Validation parser**: directly validates document if URI specified

**XML Fundamentals (Chapter 2):**

- **Filename:**
  - Can be arbitrary, document does not even need to be file at all (entry in database etc.)
  - If document is served by a web server, it will probably be assigned the MIME media type application/xml or text/xml (text/xml uses ASCII as default, which is incorrect for most, thus better use application)
- **Narrative-organized XML docs**: Mixed content of text and elements (in contrast to record-like docs where elements contain either only children or only text)

**Element:**
- Start-tag - content - end-tag
- **Empty elements**: `<element/>` equivalent to `<element> </element>`
- Case-sensitive
- Each child element (except root element, which is sometimes called **document element**) has exactly one parent, several children sometimes called **siblings**
- Each element must finish inside the element where it has started

**Attributes:**
- Name-value pair attached to start tag
- Value in single or double quotation marks
- When to put information in attribute vs when in element content?
  - Data in content, metadata as attribute
  - Additional information annotating the data are attributes

**XML names: element names and attribute names:**
- primary new feature in XML 1.1 is that XML names may contain characters only defined in Unicode 3.0 and later
- No length limit
- Can contain:
  - A - Z, a - z, 0 - 9,
  - may also include non-English letters, numbers, and ideograms, such as ö, ç, Ω, 串.
  - ’’, ’’, _
- Cannot contain:
  - Collon allowed but reserved for namespaces
  - other punctuation characters not allowed
- Can start with
  - letters
  - ideograms
  - underscore character
- Can not start with:
  - „xml“ —> reserved for standardization in W3C XML-related specifications

Entity references:
- Can only be used in element content or attribute values
- XML is markup language —> When application parses doc, it replaces reference with actual character
- Defined entity references:
  - <: &lt; or use the numeric character reference &#60;, or the hexadecimal numeric character reference &#x3C
  - >: $gt
  - &: $amp;
  - ‘: apos
- Note: only & and < must be used, the others are optional (one case when > must be escaped: sequence ]]> must be written ]]&gt;
- CDATA: Used if many < or & in text —> Everything between the <![CDATA[ and the ]]> is treated as raw character data

Comments:
- <!-- comment example -->
- The double hyphen -- must not appear anywhere inside the comment. In particular, a three-hyphen close like ---> is specifically forbidden.
- Can appear anywhere, even after or before root element, only not in tag

Processing instructions:
- In order to pass instructions to a particular application that may read the document
  - <?target ..... ?>
  - **target**: some XML name, possibly the name of the application for which processing instruction is intended
  - Rest of instruction contains **text in any syntax** (PHP for example includes large programs in processing instruction)
  - Example: <?robots index="yes" follow="no"?> in order to tell robot if it should index the page
  - Can appear anywhere in document, even before or after root
  - Processing instruction names xml, XML etc are forbidden

XML declaration:
- No processing instruction although it looks the same
- Not necessary, but if there is a XML declaration, then it must be the first thing in the
document (not even preceded by comment) because parser uses first five characters to
guess encoding
- Example: <xml version="1.0" encoding="ASCII" standalone="yes"/>
- version: use 1.0, because otherwise limited to a couple of parsers
- encoding:
  - default: UTF-8
  - If metadata is available and it conflicts with the encoding declaration, then the
    encoding specified by the metadata wins
  - standalone: if no, an application may be required to read an external DTD to determine
    proper values for parts of the doc (e.g. DTD may provide default values for attribute)
    - Docs without DTDs can have standalone=yes
    - Default: standalone=no

Well-formedness: Check for:
- Every start-tag must have a matching end-tag.
- Elements may nest but may not overlap.
- There must be exactly one root element.
- Attribute values must be quoted.
- An element may not have two attributes with the same name.
- Comments and processing instructions may not appear inside tags.
- No unescaped < or & signs may occur in the character data of an element or attribute.
- No whitespace in element tag
- Quotation marks: " ,  " or ' " '

Namespaces (Chapter 4):

- Purpose:
  - Distinguish between elements from different vocabularies that share the same name
    (Some documents combine markup from multiple XML applications)
    —> All the elements from one XML application are assigned to one URI
  - Group related elements and attributes together so that software can easily recognize
    them
- Define namespace:
  - Attach prefix —> each prefix bound to URI by xmlns:prefix attribute
  - Prefixes are bound to namespace URIs by attaching an xmlns:prefix attribute to the
    prefixed element or one of its ancestors
  - Scope: within the element where they’re declared and within its contents
  - Possible, but not recommended to redefine prefix (so it is associated to one URI in one
    element, and another in another element)
- URI:
  - URI called namespace name
  - In an XML 1.1 document, an Internationalized Resource Identifier (IRI) can be used
    instead of a URI
Namespace URIs do not necessarily point to any actual document or page (don’t even have to use the http scheme), but might be useful to place documentation at URI
- character sensitive
- Qualified name / Qname / raw name: prefix:localpart (localpart identifies element within the namespace) —> local name and namespace characterize Qname (prefix does not matter)

Syntax:
- Prefixes can be composed of any valid XML name character except the colon
- Prefix xml always bound to http://www.w3.org/XML/1998/namespace, other prefixes starting with xml are also reserved
- URI cannot be written instead of prefix directly because of / character etc, but sometimes qname is expressed with curly brackets, e.g {http://www.w3.org/1999/XSL/Transform}template (not for parser, just for human communication)

Default namespace:
- Definition: with xmlns attribute without prefix
- Then every element and child elements in that element are in default namespace
- Prefixed elements are still in their namespace, but their children are also in default
- Unprefixed attributes are never in any namespace
  (only exception: in Xml 1.1 the attribute xmlns is in a namespace)

Parser:
- Backward compatibility: parser that does not know namespaces doesn’t have any trouble reading documents with namespaces
- Parser that knows namespaces:
  - reject unmapped prefixes
  - check that only one colon in any XML name

Namespaces and DTDs:
- Completely independent, do not change DTD syntax, can be used without DTD etc
- Name of element in doc must exactly match defined name in DTD, including prefix
- Elements in Default namespace do not have prefix in DTD
- Problem: change prefix in doc —> need to change all element declarations in DTD —>
  Solution: define entity reference for prefix and colon and use this reference in element declarations

**XPATH (Chaper 9):**
- XPath: non-XML language for identifying particular parts of XML documents
- Used for example by XSLT and XPointer

Path data model:
- XPath operates on doc after entity references, CDATA sections and doc type declarations have been merged in the doc (e.g. cannot find first CDATA section with XPath)
- Can see 7 types of nodes:
  - The root node
  - Element nodes
  - Text nodes
  - Attribute nodes (xmlns and xmlns:prefix attributes are not considered attribute nodes)
  - Comment nodes
  - Processing-instruction nodes
  - Namespace nodes (rarely handled explicitly)
- Root node in XPath thus different from actual root node: root contains whole document, with children: processing instructions and comments that occur before the root node

Location path:
- Identifies set of nodes
- Built out of successive location steps, each step is evaluated relative to a particular node in the document called the context node
- Simplest location path: select root node: simple slash
- Single element name: selects all child elements of the context node
- Reference attribute with @
- Find other kind of nodes:
  - comment(): Each text node contains the maximum possible contiguous run of text not interrupted by any tag
  - processing-instruction():
    - Select all processing-instruction children of the context node
    - Specify target as argument: processing-instruction(targetName) only selects processing instructions with that target
- Each location step has two parts, axis and node test:
  - axis tells you which direction to travel from the context node
  - node test tells you which nodes to include along that axis,
  - Optional predicates further reduce the nodes according to some expression.
- Wildcards (*, node() and @*): match different elements and node types at the same time
  - asterisk *:
    - Matches any element node regardless of their name (but no comments etc)
  - prefix* —> only elements in that namespace
  - node() —> all nodes matched, also comments, attributes etc
  - @* matches all attribute nodes
- Multiple elements/attributes: | operator —> profession | hobby matches all elements of either hobby or profession
- Move around in hierarchy (like in unix filesystem):
  - . to get to root node
- .. to get to parent
- // all descendants of context node and context node itself
- Examples: //@id/.. identifies all elements in the document that have id attributes
- Predicates:
  - boolean expression to restrict set of nodes that an expression returns
  - <, >, >=, <=, !=, and, or
  - < still needs to be escaped if used inside XML doc
  - Example: //person[@id="p4567"]
  - Not necessarily boolean:
    - //name[2] selects second name element in doc (start to count from 1, not 0)
    - //name[middle_name] selects all name elements that have a child element called middle_name

Unabbreviated location paths: axis and node test separated with "::"

-> people/person/@id changes to child::people/child::person/attribute::id.
- Not used much in practice, but allows to walk along different axes than abbreviated paths (in addition to siblings, parents etc):
  - Ancestors
  - Following-sibling
  - Preceding-sibling
  - Following
  - Preceding
  - Namespace
  - Descendant
  - Ancestor-or-self

Types: (e.g. 2+2 is also a valid XPath expression, not only location paths)
- Node sets
- Numbers:
  - No integers, all numbers are 8-byte floating-point doubles
  - +, -, *, div, mod
- Strings:
  - can be compared with = and !=
- Booleans

Functions:
- No void functions
- Arguments:
  - Usually can use any type as function argument and processor will try to convert it
  - Can have variable-length argument list (e.g. concat(a, b, c ... ))
- Functions operating on node sets:
  - position(): position of current node in the context node list
  - last()
  - count()
  - id(): specify some IDs in arguments, return nodes with these IDs
'local-name()': returns local part of name of first node in the set
'namespace-uri()'
'name()': returns Qname of first node in set

String functions:
- starts-with —> e.g. starts_with("Name", "Na") would evaluate to true
- contains()
- substring-before() —> takes two string arguments and returns the substring of the first argument that precedes the initial appearance of the second argument
- substring-after()
- string-length()
- string() : convert argument to string (Booleans are converted to the string “true” or the string “false.” Node-sets are converted to the string value of the first node in the set)

Number functions
- number(): convert to number (true—>1, false—>0, string that is no number —> NaN
- round()
- floor()
- ceiling()
- sum()

Boolean functions:
- true(), false() and not()
- boolean(): converts to boolean (all numbers except NaN or zero are true, non-empty node sets are true, non-empty strings are true)
A lot of content was already covered in the lectures and exercise classes, I included ideas which were new.

Chapter 2. XML Fundamentals, [1]

- It is better to use application/xml MIME media type instead of text/xml, because the latter uses ASCII as a default.
- Some terminology:
  
  ```
  <temp>
  Lorem ipsum
  </temp>
  ```

  - **Content**: everything between start-tag and end-tag
  - **Markup**: `<temp>` and `</temp>`
  - **Character data** (text, that does not contain any tags): *Lorem ipsum* and surrounding whitespace.
- XML names (XML element names and XML attribute names) can also include non-English letters and ideograms.
  
  - XML 1.1 may use characters only defined in Unicode 3.0 and later
  - XML 1.0 may only use characters defined as of Unicode 2.0
- `]]>` should not appear in character data.
- Entity and character references (e.g. `&amp;` or `&#x03C0;`):
  
  - Can be used in: attribute values and element content
  - Cannot be used in: element names, attribute names and other markup
  - Can be used but will not be resolved in: comments and processing instructions
- CDATA section can contain anything except `]]>`.
- Comments:
  
  - cannot be closed with `--->` (one additional hyphen)
  - cannot appear inside a tag or inside another comment
- Processing instructions terminology:
  
  ```
  <?test something="yes" somethingelse="no"?>
  ```

  - **target**: test
  - **pseudo-attributes**: something and somethingelse
- XML declaration:
  
  Example:
  
  ```
  <?xml version="1.0" encoding="ASCII" standalone="yes"?>
  ```
XML declaration is not a processing instruction
- It is not mandatory, but should be first thing in the document if it is present
- `version` should be "1.0" in most cases. Version "1.1" should only be used in case if some specific languages need to be supported or if data contains obsolete control characters.
- `encoding` is not mandatory. If it is not specified, parser may try to infer it from the document. If encoding specified in declaration conflicts with encoding specified in metadata, encoding specified in metadata is used.
- `standalone` is not mandatory. If it is not specified, it is same to specified value "no". Value "no" means that an application may be required to read an external DTD. Document that has DTD can have value "yes" if DTD does not change content of the document.

Chapter 4. Namespaces, [1]
- Each prefix is mapped to a URI. With XML 1.1, IRI (Internationalized Resource Identifier) can be used instead of URI. The former enables use of non-ASCII characters. In practice, the difference is not that important since parsers do not check whether namespace names are legal URIs.
- Prefixes can contain any legal XML name character except :: also cannot be used in local name.
- Prefix `xml` is always bound to the `http://www.w3.org/XML/1998/namespace`. Prefixes beginning with any combination of upper case or lower case letters `xml` are reserved.
- In XML 1.1/Namespaces 1.1, `xmlns` attribute is in the namespace `http://www.w3.org/2000/xmlns/`, which is contrary to the fact that unprefixed attributes are not in any namespace. In XML 1.0/Namespaces 1.0 this is not the case.
- It is possible to redefine a prefix. In this case closest ancestor that defines that prefix is used.
- In XML 1.1 it is possible to undeclare a prefix.
- URIs do not need to be URLs and do not even need to use HTTP.
- If URIs differ in a single character or more, they define different namespaces (even if only one letter is upper case instead of lower case).
- Parsers:
  - Namespaces are not part of XML 1.0
  - Parser that is not aware of namespaces does not have problems with a document that contains namespaces
  - Parser that is aware of namespaces ensures:
    1. All prefixes are bound to URIs (except `xml` and `xmlns`)
    2. At most one :: is used in attribute and element names
- Namespaces and DTDs:
  - Namespaces are independent from DTDs
  - Prefixed elements and attributes must also be prefixed in DTDs
  - If prefix gets changed it has to be changed in all declarations that use the prefix in DTD. This can be solved by defining parameter entity references for prefix and ::.

Chapter 9. XPath, [1]
- XPath cannot identify CDATA sections, entity references and document type declarations, because XPath sees XML document after all three of them have already been included in the document.
- `Location path` identifies set of nodes in a document and is built out of consecutive `location steps`. Each location step is evaluated relative to `context node`.
  - `Root location path (/)` selects the root node
– **Child element location step** (e.g. `bookstore`) selects child elements of context node with name `bookstore`

– **Attribute location step** (e.g. `@category`) selects `category` attribute of context node

– **comment() location step** (comment()) selects comment node children of context node

– **text() location step** (text()) selects text node children of context node. Text nodes contain longest possible text not interrupted by a tag.


– **Wildcards:**
  * `*` matches all element nodes
  * `prefix:*` matches elements within specified namespace
  * `node()` matches all 7 kinds of nodes
  * `@*` matches all attribute nodes
  * `@prefix:*` matches all attributes within specified namespace

– **Multiple matches:** | combines multiple matches (e.g. `title|author`)

• If **Unabbreviated Location Paths** are used, axis and the node test are seperated by ::. Furthermore, eight more axes are added:

  – ancestor
  – following-sibling
  – preceding-sibling
  – following
  – preceding
  – namespace
  – descendant
  – ancestor-or-self

Examples:

– `/child::bookstore/child::book` vs. `/bookstore/book`

– `/child::bookstore/child::book[attribute::category = "COOKING"]` vs. `/bookstore/book[@category = "COOKING"]`

• **General XPath Expressions**

  – Numbers: `+`, `-`, `*`, `div`, `mod`
  – Strings: `=`, `!=`, `<`, `>`, `<=`, `>=`
  – Booleans

• **XPath Functions**

  – General:
    * No void functions
    * XPath will convert arguments such that they will match types the function expects. However, XPath cannot convert strings, booleans and numbers to node-sets.

  – Node-Set Functions:
    * `position()`: position of the current node in the context node list
    * `count()`
    * `last()`
    * `id(<string of one or more whitespace separate ids>):` node-set containing all nodes in the document with those ids
- **local-name(<node-set>):** local part (in terms of namespaces) of first node in node-set
- **namespace-uri(<node-set>):** namespace URI of first node in node-set
- **name(<node-set>):** QName of first node in node-set

- **String Functions:**
  - **string():** convert to string
  - **starts-with(<first>, <second>):** true if <first> starts with <second>
  - **contains(<first>, <second>):** true if <second> is substring of <first>
  - **substring-before(<first>, <second>):** substring of first that precedes initial appearance of <second>
  - **substring-after(<first>, <second>):** similar to previous
  - **substring(<string>, <start pos>, <length>):** if <length> omitted substring goes to the end of <string>.
  - **string-length()
  - **normalize-space(<string>):** removes extra whitespace

- **Boolean Functions:**
  - **true(), false(), not(),
  - **boolean()** evaluates to true if:
    - numbers that are not 0 or NaN
    - non-empty node-sets
    - non-empty strings

- **Number Functions**
  - **number(), round(), floor(), ceiling()
  - **sum(<node-set>):** convert each node to string and then to number and return sum of those numbers

Note that some functions can be called with or without parameters. In the latter case it is same as if context node was passed as a parameter.

**References**