1. Unit: Exercises to XML

Information about the XML course can be found at
`http://www.stud.informatik.uni-goettingen.de/xml-lecture`

Exercise 1.1 (XML-Tree vs Directory-Tree)

Load `mondial-europe.xml` into *xmllint* and browse through the directory structure. First, change into the `country` element of Germany, then into the city of Göttingen. Then, change into the next city in the document.
Links to *xmllint* and the Mondial database can be found at
`http://www.stud.informatik.uni-goettingen.de/xml-lecture`.

Exercise 1.2 (Student-DTD)

- Write a DTD for XML documents with student data:
  name, address and a student id, one or more subjects (computer science, law, chemistry, sociology etc)
- Write an XML document containing student data conforming to the DTD, and check it for validity using *xmllint*.

Exercise 1.3 (HTML-XHTML)

- Find a simple HTML document (e.g. your own personal student homepage) and convert it by hand from HTML to XHTML.
- Check the XHTML document for validity using the XHTML validator
  (`http://validator.w3.org/detailed.html`).

Hint: In your home directory in the CIP pool, there is a directory `public_html` which is your personal web directory. Files there are accessible via
`http://student.ifi.informatik.uni-goettingen.de/~`*<username>*`/`*<filename>*.

Yes, this is often a nontrivial piece of work. Most HTML pages are written very unprecise.

Why are the requirements of XML/XHTML so much stricter? The reason is the complexity of the parser: an HTML parser must be very fault-tolerant, which means that it has a lot more transitions that cover imprecise HTML.

Consider the following example of an HTML fragment (where nearly all closing tags are missing, and the table markup is far from correct):

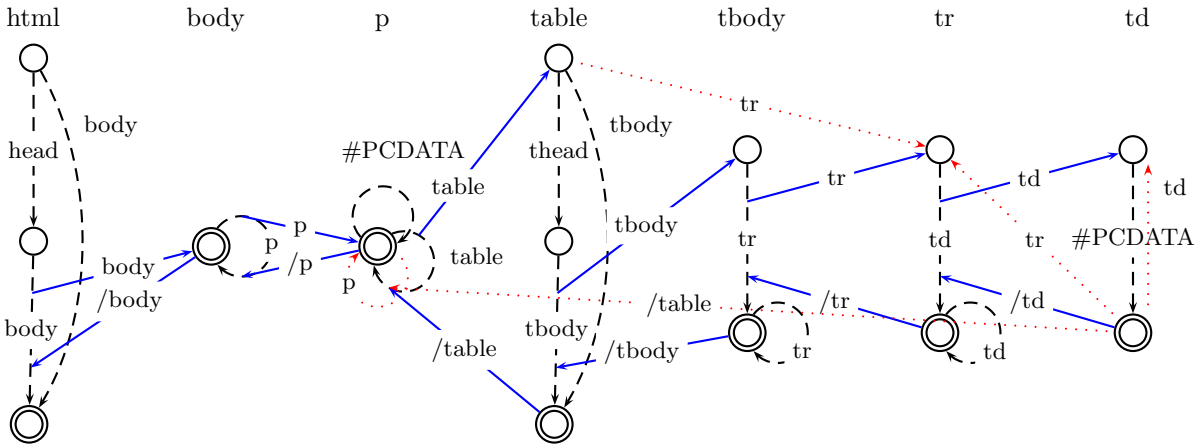
```
<html>
 <head><title>A very unprecise HTML page
 <body>
  some text
  <p>
  <table border="1">
   <tr> <td> eins.eins <td> eins.zwei
   <tr> <td> zwei.eins <td> zwei.zwei
  </table>
  <p>
  and some more text
</html>
```

Consider the following fragmentary XHTML DTD fragment:

```
<!ELEMENT html (head?,body)>
<!ELEMENT body (p*)>
<!ELEMENT p (#PCDATA|table)*>
<!ELEMENT table (thead?,tbody)>
<!ELEMENT tbody (tr+)>
<!ELEMENT tr (td+)>
<!ELEMENT td (#PCDATA)>
```

The following DFA can be used to parse and validate XHTML documents wrt. the above language fragment:

- every column is a subautomaton,
- dashed lines abstract from the structure of the subelements,
- blue lines connecting hierarchical subautomata describe transitions for opening and closing tags.



For fault-tolerant parsing of HTML, the dotted transitions must be added. They represent transitions when

- a complete level (tbody) has been omitted, or
- a closing tag has been omitted.

These lines are depicted above in red, dotted:

- opening <tr> tag in <table>: skip <tbody> level. Note that this makes the return with the closing </tr> nondeterministic – either jump back to the <tbody> level or to the <table> level (thus, the parser must push down where it came from).
- opening <td> tag in <td> element: implicitly close the </td> element and jumping to the start of a new <td>.
- opening <tr> tag in <td> element: implicitly close the </td> and the </tr>, jumping to the start of a new <tr>.
- closing <table> tag in <td> element: implicitly close the </td>, </tr>, and </tbody>, jumping to the transition that actually closes the <table>.
- opening tag in element: implicitly close the element and jumping to the start of a new .

- some more ...

These transitions cannot be defined automatically from the DTD specification, but have to be added manually by the parser designer (presuming that he knows what "shortcuts" the users will apply). Such "techniques" are in general not acceptable for any DTD – thus, for XML it was decided to have stricter rules for validation.

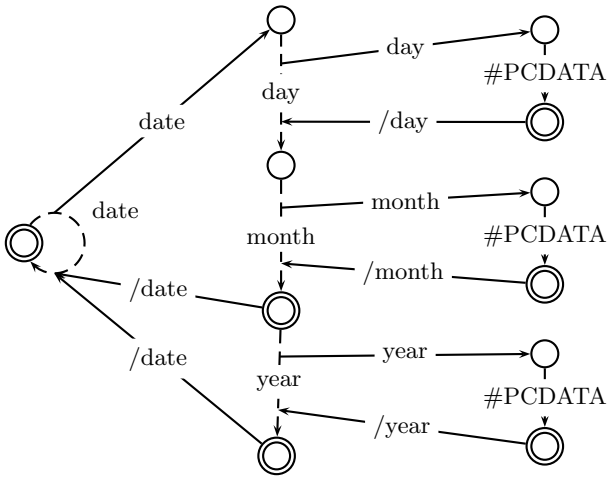
Exercise 1.4 (DFAs and DTDs)

Consider the following DTD:

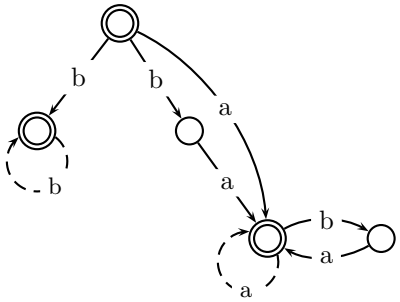
```
<!ELEMENT date (day,month,year?)>
<!ELEMENT day (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT a (date*)>
<!ELEMENT b (#PDCATA)>
<!ELEMENT c (b+|(b?,a)*)>
```

Define a finite automaton for each element definition which accepts the corresponding *content model*.

The part of the automaton for the a and date elements are simple:

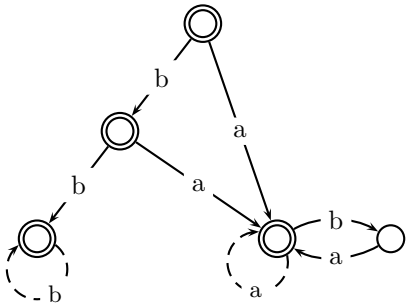


The part for the c element is mich more complicated: note that the DTD is not allowed since it is not deterministic (where determinism is defined according to the derived automaton), since if first a b is read, it is not known which branch should be entered, see below:



Note that the initial state is also an accepting state since the second branch $(b?, a)*$ can be empty.
the automaton shows that there it is nondeterministic if first, a b is read.
From other lectures, it is known that the automaton can be transformed into an equivalent deterministic one, that then also indicates an allowed content model expression.

Transformation: move the b from $b?, a$ out (take care that the second branch can also begin with an omitted b and then an a, and that it is completely optional!).



A deterministic content model expression is then
$(b, (b* | (a, (b?, a)*)))) \mid (a, (b?, a)*)?$.
Note that there are several more equivalent expressions.

The complete automaton is then created by connecting each a step with a copy of the date automaton (copies are needed since the return jump must be deterministic – in reality this is done by maintaining a stack), and connecting each b step with a #PCDATA automaton.

Exercise 1.5 (Is XML a context-free language?) Consider XML as a formal language.

(a) is the *language of all XML documents of a given document type*, specified by DTD that does not contain any attributes context-free?

(b) consider the case where the DTD contains attributes.

(c) is the *language of all well-formed XML documents*, without known document type, or with no document type at all context-free?

Recall from the Theoretical Computer Science lecture:

- A language is context-free in the sense of the Chomsky hierarchy if there is a context-free grammar for it.
- A context-free grammar's productions are of the form $A \to \alpha_1, \ldots, \alpha_n$, with A being a non-terminal symbol, and $\alpha_1, \ldots, \alpha_n$ being non-terminal and terminal symbols, $1 \leq n$.
- A common way of proving that a language is *not context-free* is to prove that the pumping lemma property does not hold.

The Pumping Lemma for context-free Languages (or $uvwxy$ Theorem). A language L is context-free $\Rightarrow \exists n \in \mathbb{N}\ \forall z \in L : |z| \geq n \to \exists uvwxy$ with $z = uvwxy$
(1) $|vx| \geq 1$
(2) $|vwx| \leq n$
(3) $\forall i \in \mathbb{N} :\ uv^i wx^i y \in L$.

In words:
For each context-free language L there is a number n such that for every word z from L of length n or longer, there is a decomposition of z into $uvwxy$, such that $(1) - (3)$ holds.

Solution. Let $\Sigma = \{<, /, >, a, \ldots, z, 0, \ldots, 9, \text{“ ”}\}$ be the (reduced) unicode alphabet of XML. A DTD consists of a number of element and attlist definitions.

(a) The grammar (recall, without attributes) is as follows:
Define L_{dtd} for a fixed DTD, which uses a fixed set $E_1, \ldots, E_n$ of element names, as a language over $\Sigma' = \{<, >, /, a, \ldots, z, 0, \ldots, 9, \text{“ ”}, \textit{elementname}_1, \ldots, \textit{elementname}_n\}$ as follows.

Starting rule: $S \to E_1 | \ldots | E_n$.

For each element definition <!ELEMENT *elementname*$_k$ EMPTY>, add the rule
$E_k \to$ <*elementname*$_k$/>.

For each element definition <!ELEMENT *elementname*$_k$ (β)>, add the rule
$E_k \rightarrow$ <*elementname*$_k$>C_β</*elementname*$_k$>.
For every C_β, add the corresponding rule for context models (= regular expressions):

if $\beta =$ `#PCDATA`	then add	$C_\beta \rightarrow P$
if $\beta =$ `ANY`	then add	$C_\beta \rightarrow E_1 \vert \ldots \vert E_n \vert \epsilon$
if $\beta = (\beta_1, \beta_2)$	then add	$C_\beta \rightarrow C_{\beta_1} C_{\beta_2}$
if $\beta = (\beta_1 \vert \beta_2)$	then add	$C_\beta \rightarrow C_{\beta_1} \vert C_{\beta_2}$
if $\beta = (\beta_1 *)$	then add	$C_\beta \rightarrow C_{\beta_1} C_\beta \vert \epsilon$
if $\beta = (\beta_1 +)$	then add	$C_\beta \rightarrow C_{\beta_1} (C_\beta \vert \epsilon)$
if $\beta = (\beta_1 ?)$	then add	$C_\beta \rightarrow C_{\beta_1} \vert \epsilon$
if $\beta =$ *elementname*$_m$	then add	$C_\beta \rightarrow A_m$,
and add the corresponding rules for C_{β_1} and C_{β_1} .		

PCDATA rule: $P \rightarrow Pa \vert Pb \vert \ldots \vert Pz \vert PA \vert \ldots \vert PZ \vert P0 \vert \ldots \vert P9 \vert P\text{“ ”} \vert \epsilon$.
The result is a context-free grammar for L_{dtd}.

(b) There are two issues to consider: (i) attributes in general, and (iii) specifics of IDREF/IDREFS.
For (ii), for every !ATTLIST *elementname*$_i$ declaration, every permutation π of the attributes *attr*$_{i,1}, \ldots,$ *attr*$_{i,m(k)}$ must be allowed in the corresponding grammar rule for E_i, e.g.
$E_k \rightarrow$ <*elementname*$_k$ *attr*$_{k,\pi(1)}$=“P” $\ldots$ *attr*$_{k,\pi(m(k))}$=“P”>C_β</*elementname*$_k$>.
If instead of only CDATA attributes, also NMTOKEN/NMTOKENS are allowed, use a rule for NM and NMS similar to P.
So far, the grammar is still context-free.
If ID/IDREFS with their uniqueness and reference properties have to be verified, these features go beyond the expressiveness of context-free grammars. Actually, the context-free grammar together with a dictionary (to store IDs and check their uniqueness) and a postprocessing run (to check IDREFS that may be forward-references) is used.

(c) Let $L_{xml} = \{$ <$\omega_1 \ldots \omega_k$>$\alpha_1 \ldots \alpha_m$</$\omega_1 \ldots \omega_k$>$\vert$,
$\omega_1 \in \{a, \ldots, z, A, \ldots, Z\}$,
$\omega_2, \ldots, \omega_k \in \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}$,
$\alpha_1, \ldots, \alpha_m \in \{\epsilon\} \cup$ `CHAR` $\cup L\}$
(well-formed XML without document type and without attributes).
Show via pumping lemma that L_{xml} is not context-free.
To prove: $\forall n \in \mathbb{N}\ \exists z \in L, \vert z \vert \geq n\ \forall uvwxy$ with $z = uvwxy$:
$(1):\ \vert vx \vert = 0\ \vee\ (2):\ \vert vwx \vert > n\ \vee (3):\ \exists i \in \mathbb{N}:\ uv^i wx^i y \notin L$.
Let $z =< \underbrace{a \ldots a}_{n\times} \underbrace{b \ldots b}_{n\times} >< / \underbrace{a \ldots a}_{n\times} \underbrace{b \ldots b}_{n\times} >$.
Assuming $\vert vx \vert \geq 1 \wedge \vert vwx \vert \leq n :\Rightarrow$
case 1 : $vwx \subseteq$ starting tag or $vwx \subseteq$ end tag : $\Rightarrow\ uv^2wx^2y \notin L$.
case 2 : $v \cap \{>, <, /\} \neq \emptyset$ or $x \cap \{>, <, /\} \neq \emptyset$: $\Rightarrow\ uv^2wx^2y \notin L$.
case 3 : $v \subseteq$ starting tag $\wedge\ x \subseteq$ end tag :
$\vert vwx \vert \leq n \Rightarrow v = b \ldots b, x = a \ldots a \Rightarrow$
$uvwxy = \underbrace{< a \ldots a}_{u} \underbrace{b \ldots b}_{v} \underbrace{b \ldots b}_{w} \underbrace{>< / a \ldots a}_{x} \underbrace{a \ldots a}_{y} \underbrace{a \ldots a b \ldots b >}_{} \Rightarrow$
$uv^2wx^2y =< \underbrace{a \ldots a}_{n} \underbrace{b \ldots b}_{n+\vert v \vert} >< / \underbrace{a \ldots a}_{n+\vert x \vert} \underbrace{b \ldots b}_{n} > \notin L_{xml}$.

Comments. Thus, parsing an XML document is much easier if a DTD is provided.

If an XML document is processed, where no DTD is provided, the parsing is theoretically based on generating a context-free grammar on-the-fly by instantiating the ANY rule for every element

name that is unknown so far.

Actually, this is again encoded in a stream processing that reads the plain text input character by character and determines element names, attribute names, values etc., and does the parsing (and often already the intended processing).

Exercise 1.6 (XML Tree and XPath Axes) Consider the XPath axes in a document.

- Provide equivalent characterizations of the "following" axis and of the "preceding" axis
  i) in terms of "preorder" and "postorder",
  ii) in terms of other axes.

Given an XML document, let $pre : element \cup text \to \mathbb{N}$ and $post : element \cup text \to \mathbb{N}$ denote the preorder and postorder numberings.

- Given a node x, $preceding(x)$ are all elements and text nodes y such that $pre(y) < pre(x)$ and not $post(y) < post(x)$ (this excludes the ancestors). Enumerate them in reverse document order.
- Symmetric: Given a node x, $following(x)$ are all elements and text nodes y such that $post(y) > post(x)$ (note that this does not include the descendants, since the root of a subtree is visited after the tree) and not $pre(y) < pre(x)$ (this excludes the ancestors). Enumerate them in document order.
- The nodes on the "following" axis can be enumerated as "descendants-or-self of all following siblings of the ancestors-or-self of x".
  XPath Expression:
  `doc('mondial.xml')//city[name='Karlsruhe']/ancestor-or-self::*/following-sibling::*/descendant-or-self::*/name()`
  Analogously for the "preceding" axis.

Exercise 1.7 (XML to RDB) A possible model for storing (or indexing) XML data is based on relational tables (we ignore namespaces here).

(1) a table for storing element and text nodes:
  - first column: node identifier in Dewey Notation (e.g., 1.2.6.3 for the third child of the sixth child of the second child of the root node),
  - second column: number of the node when enumerated in *preorder*,
  - third column: number of the node when enumerated in *postorder*,
  - forth column: element type (or "text"),
  - fifth column: text content (or NULL).

(2) a table for storing attribute nodes:
  - first column: dewey identifier of the node where the attribute belongs to,
  - second column: attribute name,
  - third column: value.

a) Discuss whether the above information is sufficient for storing an XML document. Give the tables for a small example document.

b) Discuss what must be done when an update (modification, insertion, deletion) is executed.

c) Given a "current" element somewhere in the tree, characterize the following sets of nodes (i.e., the nodes that result from navigating along the different axes) by their dewey notation and, if possible, by their preorder / postorder information:
  - the parent
  - all children

- all successors
- all ancestors
- all siblings
- all predecessors according to document order
- all successors according to document order
- all attributes

Consider the following XML tree (also available on the Web page):

```
<mondial>
 <country car_code="F" area="547030" capital="cty-France-Paris">
  <name>France</name>
  <population>58317450</population>
  <population_growth>0.34</population_growth>
  <languages percentage="100">French</languages>
  <province capital="cty-France-Strasbourg">
   <name>Alsace</name>
   <city id="cty-France-Strasbourg">
    <name>Strasbourg</name>
    <population year="90">252338</population>
   </city>
   <city>
    <name>Mulhouse</name>
    <population year="90">108357</population>
   </city>
  </province>
  <province capital="cty-France-Paris">
   <name>Ile de France</name>
   <city id="cty-France-Paris">
    <name>Paris</name>
    <population year="90">2152423</population>
   </city>
   :
  </province>
  :
 </country>
 <country car_code="D" area="356910" capital="cty-Germany-Berlin">
  <name>Germany</name>
  <population>83536115</population>
  <population_growth>0.67</population_growth>
  <languages percentage="100">German</languages>
  <province>
   <name>Baden Wurttemberg</name>
   <city>
    <name>Stuttgart</name>
    <population year="95">588482</population>
   </city>
   <city>
    <name>Karlsruhe</name>
    <population year="95">277011</population>
   </city>
```

```
  </province>
  :
  <province>
   <name>Berlin</name>
   <city id="cty-Germany-Berlin">
    <name>Berlin</name>
    <population year="95">3472009</population>
   </city>
  </province>
  :
 </country>
 <country car_code="H" area="93030" capital="cty-Hungary-Budapest">...</country>
 :
</mondial>
```

The resulting table is as follows:

```
Elements:
Dewey Nr     Element type| text contents |preorder|postorder

1            mondial                        1
1.1          country                        2    150
1.1.1        name                           3      2
1.1.1.1                  "France"           4      1
1.1.2        population                     5      4
1.1.2.1                  58317450           6      3
1.1.3        population_growth              7      6
1.1.3.1                  0.34               8      5
1.1.4        languages                      9      8
1.1.3.1                  "French"          10      7
1.1.5        province                      11     21
1.1.5.1      name                          12     10
1.1.5.1.1                "Alsace"          13      9
1.1.5.2      city                          14     15
1.1.5.2.1    name                          15     12
1.1.5.2.1.1              "Strasbourg"      16     11
1.1.5.2.2    population                    17     14
1.1.5.2.2.1              252338            18     13
1.1.5.3      city                          19     20
1.1.5.3.1    name                          20     17
1.1.5.3.1.1              "Mulhouse"        21     16
1.1.5.3.2    population                    22     19
1.1.5.3.2.1              108357            23     18
1.1.6        province                      24     29
1.1.6.1      name                          25     23
1.1.6.1.1                "Ile de France"   26     22
1.1.6.2      city                          27     28
1.1.6.2.1    name                          28     25
1.1.6.2.1.1              "Paris"           29     24
1.1.6.2.2    population                    30     27
1.1.6.2.2.1              2152423           31     26
  :
```

```
Note: we assume that Germany is node No 152 in preorder
enumeration.
That means, that Node no.1 is ‘‘mondial’’ and France
consists of 150 nodes (including the country node for it).
Thus, in postorder enumeration, the country node for
France has number 150. The first node in postorder in
the Germany subtree, the text contents of the name, has
number 151.

1.2          country                        152
1.2.1        name                           153    152
1.2.1.1                   "Germany"         154    151
1.2.2        population                     155    154
1.2.2.1                   83536115          156    153
1.2.3        population_growth              157    156
1.2.3.1                   0.67              158    155
1.2.4        languages                      159    158
1.2.4.1                   "German"          160    157
1.2.5        province                       161    171
1.2.5.1      name                           162    160
1.2.5.1.1                 "Baden Wurttemberg" 163 159
1.2.5.2      city                           164    170
1.2.5.2.1    name                           165    162
1.2.5.2.1.1               "Stuttgart"       166    161
1.2.5.2.2    population                     167    164
1.2.5.2.2.1               588482            168    163
1.2.5.3      city                           169    169
1.2.5.3.1    name                           170    166
1.2.5.3.1.1               "Karlsruhe"       171    165
1.2.5.3.2    population                     172    168
1.2.5.3.2.1               277011            173    167
1.2.6        province                       174      ?
  :
1.2.7        province                       210
1.2.7.1      name                           211
1.2.7.1.1                 "Berlin"          212
1.2.7.2      city                           213
1.2.7.2.1    name                           214
1.2.7.2.1.1               "Berlin"          215
1.2.7.2.2    population                     216
1.2.7.2.2.1               3472009           217
  :
1.3          country                        389      ?
1.3.1        name                           390    389
1.2.1.1                   "Hungary"         391    388
 :

Attributes:

Parent(Dewey) AttrName Attr Value
1.1         car_code    "F"
```

```
1.1         area        "547030"
1.1         capital     "cty-France-Paris"
1.1.4       percentage  "100"
1.1.5       capital     "cty-France-Strasbourg"
1.1.5.2     id          "cty-France-Strasbourg"
1.1.5.2.2   year        "90"
1.1.6       capital     "cty-France-Paris"
1.1.6.2     id          "cty-France-Paris"
1.1.6.2.2   year        "90"
1.2         car_code    "D"
1.2         area        "356910"
1.2         capital     "cty-Germany-Berlin"
etc.
```

The information is more than sufficient: The *preorder* and *postorder* numbers are not necessary. But they will provide useful search indexes.

Note that there is no reasonable notion for *inorder* traversal (this would be "leftchild-self-rightchild" an is thus only applicable to *binary* trees).

Updates:

- update of text contents: only one update of the first table
- modification, insertion, or deletion of an attribute node: only one update to the second table
- insertion or deletion of an element:
  - change dewey number of all following siblings
  - change preorder and postorder numbers of all nodes with higher numbers

Use of the indexes:

- parent, following-sibling, preceding-sibling: by Dewey Number arithmetics (note that `CREATE TYPE DEWEY` with suitable methods `parent()`, `preceding-sibling()`, `following-sibling()` and an `ORDER` method makes this even easier [note that there cannot be a `MAP` method if the number of children of a node is not restricted]). Use also an index on this column.
- descendants: all nodes x with `self.preorder <` x`.preorder and self.postorder >` x`.postorder`
- children: descendants+Dewey comparison, or add a `depth` column or `depth` function to the Dewey type.
- ancestors: all nodes x with `self.preorder >` x`.preorder and self.postorder <` x`.postorder`.
- following: all nodes x with `self.preorder <` x`.preorder` and `self.postorder <` x`.postorder` (neither ancestors nor descendants are following).
- preceding: all nodes x with `self.postorder >` x`.postorder` and `self.preorder >` x`.preorder` (note: following and preceding do not include the ancestors, but only nodes that are the roots of trees that *completely* follow/precede `self`!)

Optimizations:

- "gaps" in the preorder or postorder numbering reduce update efforts (since both are only used for comparisons, that does not matter in most cases)
- use relative numbers wrt. the previous sibling or the parent (amortized analysis!). Note that $post(x) = pre(x) - depth(x) + number - of - descendants(x)$
  Proof: when a node is enumerated in postorder, the following nodes have been enumerated before: all "preceding"nodes in preorder except the ancestors on the way back to the root, additionally, all nodes in the subtree rooted in x.

- Thus, if for each node, the size of the subtree rooted in it is known, $pre(x)$ and $post(x)$ can be computed as follows:
  - $pre(x)$ = sum of sizes of all subtrees that are rooted in preceding siblings of x's ancestors + #(ancestors), and
  - $post(x)$ = sum of sizes of all subtrees that are rooted in preceding siblings of x's ancestors + sum of sizes of the tree rooted in x - 1.

Further exercise (solutions to be sent to us):

- create suitable tables in SQL, including a Dewey Object Type,
- implement an XSLT stylesheet or a recursive XQuery function or a recursive OraXML PL/SQL function (see later) that traverses an XML tree and creates suitable input statements,
- experiment with SQL queries for the axes.