1. Versuch: Aufgaben zu XML

`http://www.stud.informatik.uni-goettingen.de/xml-lecture`

Aufgabe 1.1 (XML-Tree vs Directory-Tree)

Laden Sie `mondial-europe.xml` in *xmllint* und schauen Sie sich die Ordnerstruktur an. Wechseln Sie erst ins Land Deutschland, und dann in die Stadt Göttingen. Wechseln Sie jetzt zur übernächsten Stadt im Dokument.
Verweise auf *xmllint* und die Mondial-Dateien finden Sie unter
`http://www.stud.informatik.uni-goettingen.de/xml-lecture` `http://www.stud.informatik.uni-goettingen.de/xml-lecture`.

Aufgabe 1.2 (Student-DTD)

- Schreiben Sie eine DTD für XML-Dokumente mit Studentendaten:
  Ein Student hat einen Namen, eine Adresse, eine Matrikelnummer, ein oder mehrere Studienfächer usw.
- Schreiben Sie zu der DTD ein XML-Dokument, das Studentendaten enthält, und validieren Sie diese Beispielinstanz mit *xmllint*.

Aufgabe 1.3 (HTML-XHTML)

- Suchen Sie sich ein einfaches HTML-Dokument (z.B. Ihre eigene Studierenden-Homepage) und wandeln Sie sie von Hand von HTML nach XHTML um.
- Validieren Sie die Seite mit dem XHTML-Validator, der unter
  (`http://validator.w3.org/detailed.html`). zu finden ist.

Hinweis: In Ihrem Home-Verzeichnis im CIP-Pool unter
`http://stud.ifi.informatik.uni-goettingen.de/` ∼⟨*username*⟩ existiert ein Verzeichnis `public_html`.
Dies ist Ihr Webverzeichnis. Auf Seiten, die Sie dort ablegen, können Sie über
`http://stud.ifi.informatik.uni-goettingen.de/`∼⟨*username*⟩`/`⟨*dateiname*⟩ per Browser zugreifen.

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):

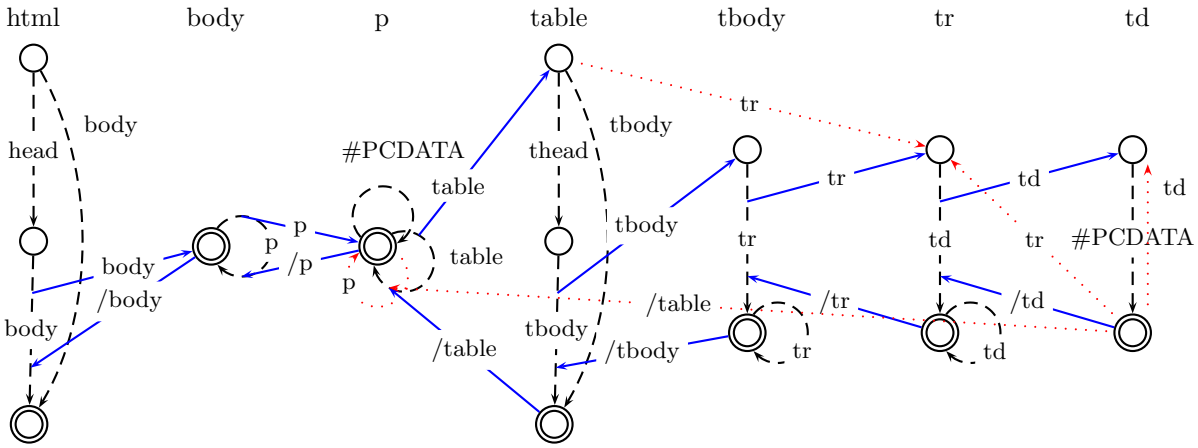
```
<html>
 <head><title>A very unprecise HTML page
 <body>
  some text
  <p>
  <table border="1">
   <tr> <td> eins.eins <td> eins.zwei
   <tr> <td> zwei.eins <td> zwei.zwei
  </table>
  <p>
  and some more text
</html>
```

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 ...

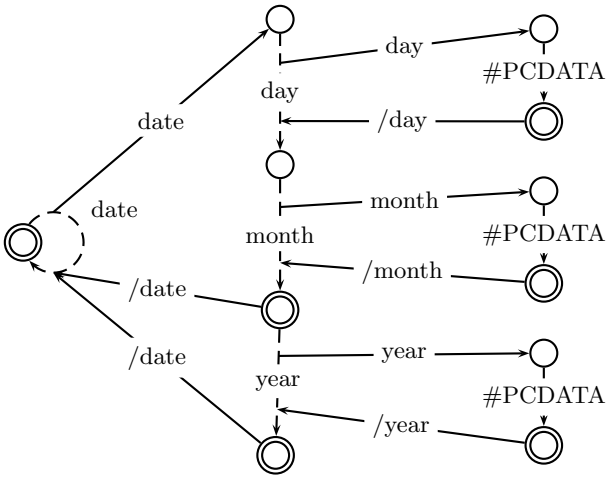
These transitions cannot be defined automatically from the DTD specification, but have to be added manually by the parser designer (presuming that he knows what "shortcuts" the users will apply). Such "techniques" are in general not acceptable for any DTD – thus, for XML it was decided to have stricter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

Aufgabe 1.4 (DFAs and DTDs) Gegeben sei folgende DTD:

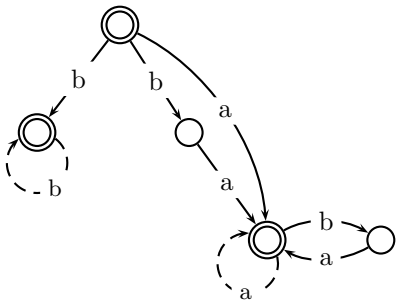
```
<!ELEMENT date (day,month,year?)>
<!ELEMENT day (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT a (date*)>
<!ELEMENT b (#PDCATA)>
<!ELEMENT c (b+|(b?,a)*)>
```

Geben Sie zu jeder Elementdefinition einen deterministischen endlichen Automaten an, der das jeweilige Content Model akzeptiert.
Hinweis: Starten Sie mit den Automaten für die einfachen Content Models und bauen Sie die komplizierteren Automaten aus den einfachen zusammen.

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

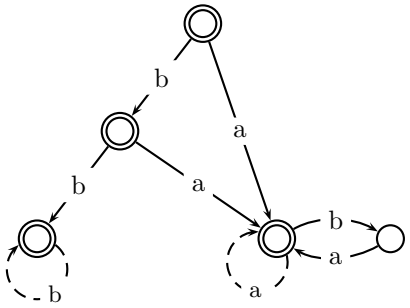


The part for the `c` element is mich more complicated: note that the DTD is not allowed since it is not deterministic (where determinism is defined according to the derived automaton), since if first a `b` is read, it is not known which branch should be entered, see below:



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.

Aufgabe 1.5 (Is XML a contextfree language?) 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 $\Rightarrow \exists n \in \mathbb{N}\ \forall z \in L, |z| \geq n\ \exists uvwxy$ 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 exists a number n such that for each 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 (ignoring all attributes) is as follows:
Let L_{xml} for a fixed DTD, which yieldas a fixed set of element names, be a language over $\Sigma' = \{<, >, /, a, \ldots, z, 0, \ldots, 9, \text{“ ”}, \mathit{elementname}_1, \ldots, \mathit{elementname}_n\}$.

Starting rule: $S \to X$.
PCDATA rule: $P \to Pa|Pb|\ldots|Pz|PA|\ldots|PZ|P0|\ldots|P9|P\text{“ ”}|\epsilon$.
For each element definition <!ELEMENT *elementname*$_k$ EMPTY>, add the rule
$A_k \to$ <*elementname*$_k$/>.
For each element definition <!ELEMENT *elementname*$_k$ (β)>, add the rules
$A_k \to$ <*elementname*$_k$>X_β</*elementname*$_k$> and

if $\beta =$ `#PCDATA`	then add	$X_\beta \to P$
if $\beta =$ `EMPTY`	then add	$X_\beta \to \epsilon$
if $\beta =$ `ANY`	then add	$X_\beta \to (X_\beta(A_1 \mid \ldots \mid A_n))\mid\epsilon$
if $\beta = (\beta_1, \beta_2)$	then add	$X_\beta \to X_{\beta_1} X_{\beta_2}$
if $\beta = (\beta_1 \mid \beta_2)$	then add	$X_\beta \to X_{\beta_1} \mid X_{\beta_2}$
if $\beta = (\beta_1 *)$	then add	$X_\beta \to (X_{\beta_1} X_\beta)\mid\epsilon$
if $\beta = (\beta_1 +)$	then add	$X_\beta \to X_{\beta_1}((X_{\beta_1} X_\beta)\mid\epsilon)$
if $\beta = (\beta_1 ?)$	then add	$X_\beta \to X_{\beta_1}\mid\epsilon$
if $\beta = elementname_m$	then add	$X_\beta \to A_m$.

Finally, add for each $elementname_k$ the rule $X \to A_k$. The result is a context-free grammar for L_{xml}.

(b) Let $L = \{$ $<\omega_1 \ldots \omega_k>\alpha_1 \ldots \alpha_m</\omega_1 \ldots \omega_k>\mid$,
$\omega_1 \in \{a, \ldots, z, A, \ldots, Z\}$,
$\omega_2, \ldots, \omega_k \in \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9\}$,
$\alpha_1, \ldots, \alpha_m \in \{\epsilon\} \cup$ `CHAR` $\cup L\}$

(well-formed XML without document type and without attributes).

Show via pumping lemma that L is not context-free.

To prove: $\forall n \in \mathbb{N}\ \exists z \in L, |z| \geq n\ \forall uvwxy$ with $z = uvwxy$:
$(1):\ |vx| = 0\ \vee\ (2):\ |vwx| > n\ \vee (3):\ \exists i \in \mathbb{N}:\ uv^i wx^i y \notin L.$
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 $|vx| \geq 1 \wedge |vwx| \leq n :\Rightarrow$

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 :

$|vwx| \leq n \Rightarrow v = b \ldots b, x = a \ldots a \Rightarrow$

$uvwxy = \underbrace{< a \ldots a b \ldots b}_{u}\underbrace{b \ldots b}_{v}\underbrace{b \ldots b >< / a \ldots a}_{w}\underbrace{a \ldots a}_{x}\underbrace{a \ldots a b \ldots b >}_{y} \Rightarrow$

$uv^2wx^2y =< \underbrace{a \ldots a}_{n}\underbrace{b \ldots b}_{n+|v|} >< / \underbrace{a \ldots a}_{n+|x|}\underbrace{b \ldots b}_{n} > \notin L$

.

Aufgabe 1.6 (XML Tree and XPath 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.

Aufgabe 1.7 (XML to RDB)

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.