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, or any (simple) page from the Web) 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; see Exercise 1.4

Exercise 1.4 (HTML-XHTML)

- Consider the following fragmentary XHTML DTD fragment :

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

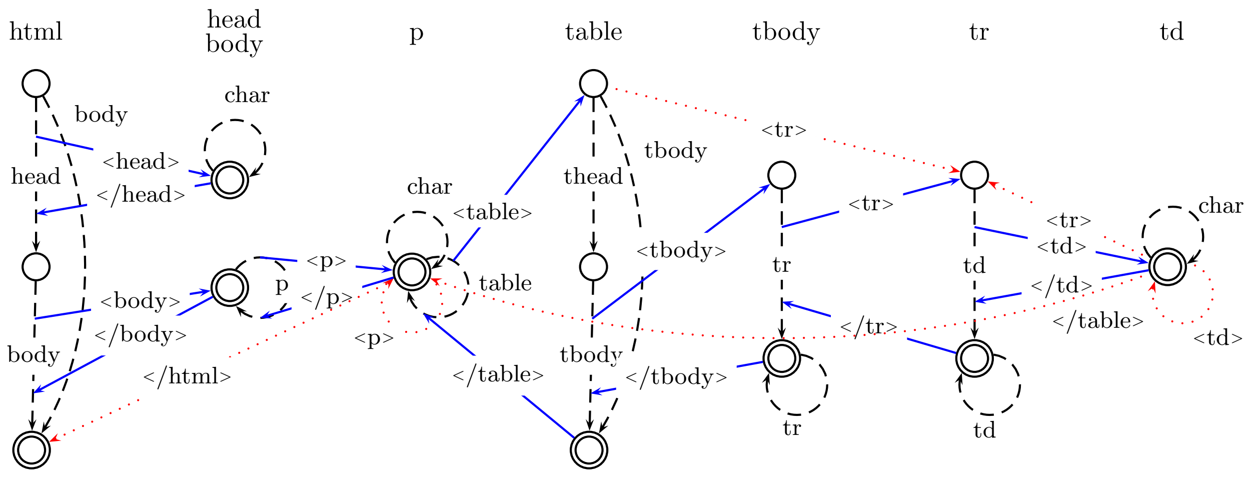
  Give a DFA that accepts the language of that DTD.
- Consider the following example of an HTML fragment (where nearly all closing tags are missing, and the table markup is far from correct):

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

Extend your DFA such that it accepts this fragment.

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.



Simplifications:

- The sub-sutomaton for body must be duplicated for the transition where <html> contains no head.
- The sub-automaton for thead is omitted (it is also not listed in the DTD).
- The sub-sutomaton for tbody must be duplicated for the transition where <table> contains no thead.
- The sub-automata for tr, and td must be duplicated for the cycle transitions in the accepting states of the tbody and tr sub-automata.
- note: black transitions are labeled with the element names (they represent “complex” transitions inside each level-automaton), while blue (and red, see below) transitions are labeled with the actual opening and closing tags.

In an implementation, the duplication is replaced by “calling” a subautomaton and maintaining a return stack.

For fault-tolerant parsing of HTML, the (red) 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 <p> tag in <p> element: implicitly close the </p> element and jumping to the start of a new <p>.
- 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.

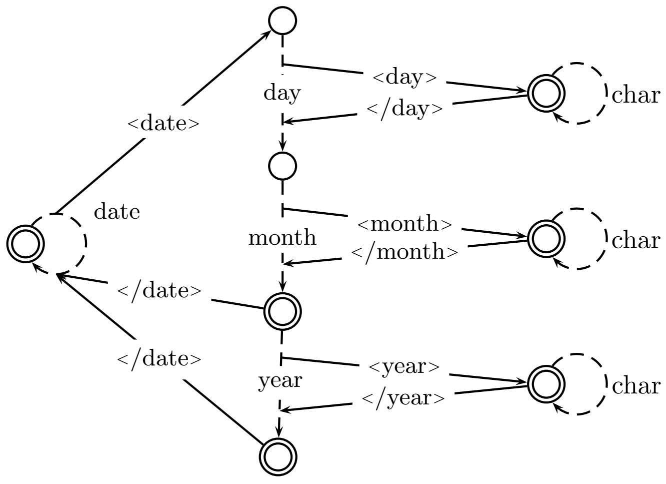
Exercise 1.5 (DFAs and DTDs)

Consider the following DTD:

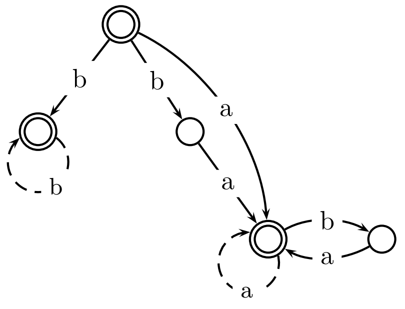
```
<!ELEMENT date (day,month,year?)>
<!ELEMENT day (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT a (date*)>
<!ELEMENT b (#PDCATA)>
<!ELEMENT c (b+|(b?,a)*)>
```

Give a deterministic finite automaton for each element definition which accepts the corresponding *content model* and connect the automata to parse XML files acoording to this DTD.

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

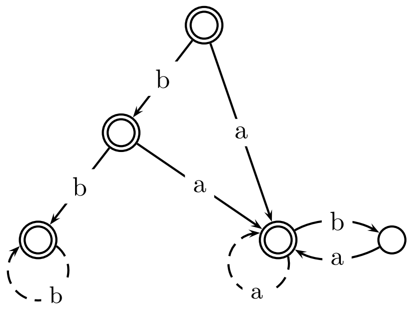


The part for the c element is much more complicated: note that the DTD is not allowed since it is not deterministic (where determinism is defined according to the derived automaton), since if first a b is read, it is not known which branch should be entered, see below:



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 1.6 (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 a 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 $A \to \alpha_1, \dots, \alpha_n$, with A being a non-terminal symbol, and $\alpha_1, \dots, \alpha_n$ 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). For each context-free language L there is a number $n \in \mathbb{N}$ such that for all words $z \in L$ of length n or longer, there is some decomposition $z = uvwxy$ such that
(1) $|vx| \geq 1$
(2) $|vwx| \leq n$
(3) $\forall i \in \mathbb{N}: \ uv^iwx^iy \in L$.

Solution. Let $\Sigma = \{<, /, >, a, \dots, z, 0, \dots, 9, \text{“ ”}\}$ 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 $E_1, \dots, E_n$ of element names, as a language over $\Sigma' = \{<, >, /, a, \dots, z, 0, \dots, 9, \text{“ ”}, \textit{elementname}_1, \dots, \textit{elementname}_n\}$ as follows.

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

For each element definition <!ELEMENT *elementname*$_k$ EMPTY>, add the rule
$E_k \to$ <*elementname*$_k$/>.
For each other element definition that uses a non-empty content model <!ELEMENT *elementname*$_k$ β>, add the rule
$E_k \to$ <*elementname*$_k$>C_β</*elementname*$_k$>
(C_β is a new non-terminal that produces expressions according to the content model β).
For every such C_β, add the corresponding rule for context models (= regular expressions):

if $\beta =$ `(#PCDATA)`	then add $C_\beta \to P$
if $\beta =$ `ANY`	then add $C_\beta \to E_1 C_\beta | \dots | E_n C_\beta | \epsilon$
if $\beta = (\beta_1, \beta_2)$	then add $C_\beta \to C_{\beta_1} C_{\beta_2}$
if $\beta = (\beta_1 | \beta_2)$	then add $C_\beta \to C_{\beta_1} | C_{\beta_2}$
if $\beta = (\beta_1 *)$	then add $C_\beta \to C_{\beta_1} C_\beta | \epsilon$
if $\beta = (\beta_1 +)$	then add $C_\beta \to C_{\beta_1} (C_\beta | \epsilon)$
if $\beta = (\beta_1 ?)$	then add $C_\beta \to C_{\beta_1} | \epsilon$
if $\beta =$ *elementname*$_m$	then add $C_\beta \to E_m$,

and recursively add the corresponding rules for C_{β_1} and C_{β_2} .

PCDATA rule: $P \to \varepsilon | \text{“a”}P | \text{“b”}P | \dots | \text{“z”}P | \text{“A”}P | \dots | \text{“Z”}P | \text{“0”}P | \dots | \text{“9”}P | \text{“ ”}P | \dots$
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*$_i$ *attr*$_{i,1}$ ...*attr*$_{i,m(i)}$ > declaration, every permutation π of the attributes *attr*$_{i,1}, \dots,$ *attr*$_{i,m(i)}$ must be allowed in the corresponding grammar rule for E_i, e.g.
$E_k \to$ <*elementname*$_k$ *attr*$_{k,\pi(1)}$=“P” ... *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* (without whitespace) 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 $L_{xml} = \{ \texttt{<}\omega_1 \ldots \omega_k\texttt{>}\alpha_1 \ldots \alpha_m\texttt{</}\omega_1 \ldots \omega_k\texttt{>} \mid$,
$\omega_1 \in \{a, \ldots, z, A, \ldots, Z\}$,
$\omega_2, \ldots, \omega_k \in \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \text{␣}, \ldots\}$,
$\alpha_1, \ldots, \alpha_m \in \{\varepsilon\} \cup \texttt{CHAR} \cup L_{xml}\}$

(well-formed XML without attributes, note the recursive definition of L_{xml}).

Show the via pumping lemma that L_{xml} is not context-free.

To prove: For all $n \in \mathbb{N}$, there is a word $z \in L_{xml}$ with length $|z| \geq n$, for all decompositions $z = uvwxy$:
(1): $|vx| = 0$ or (2): $|vwx| > n$ or (3): $\exists i \in \mathbb{N}: \ uv^iwx^iy \notin L_{xml}$.

As usual for such proofs, one does not consider the most general case, but a carefully chosen (usually quite simple) word:

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 $|vx| \geq 1$ and $|vwx| \leq n$, there are three possible cases:

– $vwx \subseteq$ opening tag or $vwx \subseteq$ closing tag $:\Rightarrow uv^2wx^2y \notin L$.
(duplication takes place either inside the opening tag or inside the closing tag $\rightarrow$ do not match any more)

– $v \cap \{>, <, /\} \neq \emptyset$ or $x \cap \{>, <, /\} \neq \emptyset :\Rightarrow uv^2wx^2y \notin L$.
(one of v or x duplicates inside the opening or closing tag, the other one duplicates something with the "></")

– $v \subseteq$ opening tag $\wedge\ x \subseteq$ closing tag:
$|vwx| \leq n \Rightarrow v = b \ldots b, x = a \ldots a \Rightarrow$
$uvwxy = \underbrace{< a \ldots ab \ldots b}_{u}\underbrace{b \ldots b}_{v}\underbrace{b \ldots b >< / a \ldots a}_{w}\underbrace{a \ldots a}_{x}\underbrace{a \ldots ab \ldots b >}_{y} \Rightarrow$
$uv^2wx^2y = < \underbrace{a \ldots a}_{n}\underbrace{b \ldots b}_{n+|v|} >< / \underbrace{a \ldots a}_{n+|x|}\underbrace{b \ldots b}_{n} > \notin L_{xml}$.

Comments on the Pumping Lemma Proof. The sample instance of the proof is chosen carefully: a simple instance. This serves not primarily to have a simple proof, but to know enough about the instance to show that *every* split into vwx yields a word not in L_{xml}.

What happens, if it is known that the length of element names is constrained by some maximum m?
The proof fails, because it works only for $n \leq m/2$. Note that this only means that this Pumping Lemma proof fails, but yet nothing about $L_{xml/max}$ in general. Maybe there is another Pumping Lemma proof? Or is $L_{xml/max}$ context-free?

Actually, under this restriction it is context-free. A grammar can obviously be given by instantiating the rule for elements for any m-letter string *abc* that is allowed for element names:

$S \rightarrow$ <*abc*>S</*abc*> for every m-letter string *abc* that is allowed for element names
$S \rightarrow P \mid \epsilon$. ($P$ for characters as before)

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

This theoretical consideration has important practical consequences: scanning any XML document once for the used element names is sufficient to instantiate an appropriate context-free grammar (i.e., using exactly these element names for *abc* above).

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