1. Unit: Warm-up Exercises

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

- 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):

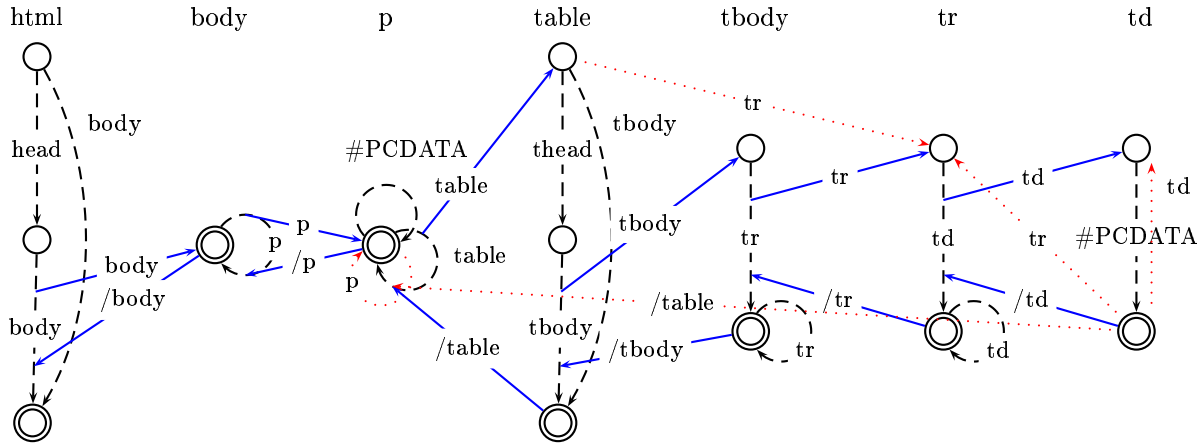
```
<html>
 <head><title>A very unprecise HTML page
 <body>
  some text
  <p>
  <table border="1">
   <tr> <td> eins.eins <td> eins.zwei
   <tr> <td> zwei.eins <td> zwei.zwei
  </table>
  <p>
  and some more text
</html>
```

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.

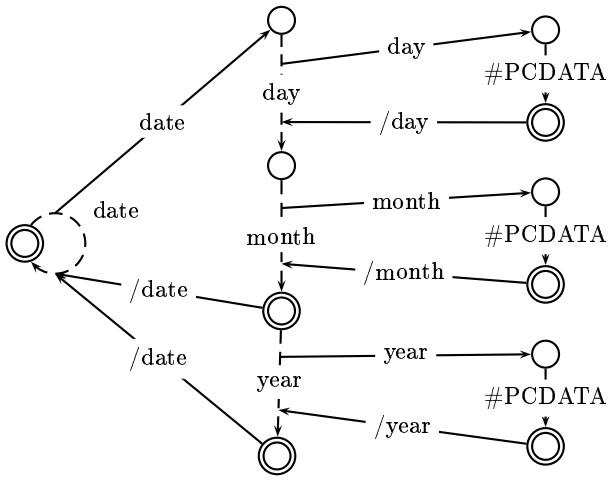
Exercise 1.4 (DFAs and DTDs)

Consider the following DTD:

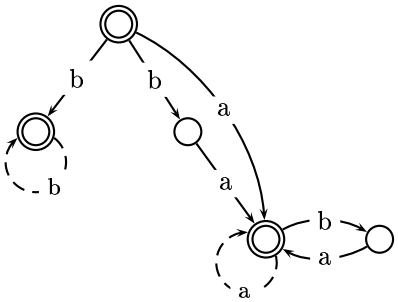
```
<!ELEMENT date (day,month,year?)>
<!ELEMENT day (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT a (date*)>
<!ELEMENT b (#PDCATA)>
<!ELEMENT c (b+|(b?,a)*)>
```

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:

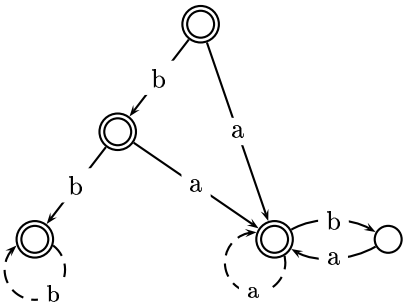


The part for the c element is mich more complicated: note that the DTD is not allowed since it is not deterministic (where determinism is defined according to the derived automaton), since if first a b is read, it is not known which branch should be entered, see below:



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 determninistic – in reality this is done by maintaining a stack), and connecting each b step with a #PCDATA automaton.