

Part VI

Valid XML—DTDs and XML Schema

Outline of this part

- 1 Valid XML
- 2 DTDs—Document Type Definitions
 - Element Declaration
 - Attribute Declaration
 - Crossreferencing via ID and IDREF
 - Other DTD Features
 - A “Real Life” DTD—GraphML
 - Concluding remarks on DTDs
- 3 XML Schema
 - Some XML Schema Constructs
 - Other XML Schema Concepts
- 4 Validating XML Documents Against DTDs
 - Regular Expressions
 - Evaluating Regular Expressions (Matching)
 - Plugging It All Together

Valid XML

- More often than not, applications that operate on XML data require the XML input data to conform to a **specific XML dialect**.



- This requirement is more strict than just XML well-formedness.
- The (hard-coded) application logic relies on, *e.g.*,
 - the **presence** or absence of specifically named elements [attributes],
 - the **order** of child elements within an enclosing element,
 - attributes having exactly one of several **expected values**, ...
- If the input data fails to meet the requirements, results are often disastrous.

Example: Transform element amount into attribute:

```
<bet gambler="doe"><amount>7</amount>...</bet>
```



```
<bet gambler="doe" amount="7">...</bet>
```

Stumbling Code

```

1  $ java foo input.xml
2  Calculate gambling results.....
3  Exception in thread "main" java.lang.NumberFormat
4      at java.lang.Integer.parseInt(Integer.java:394)
5      at java.lang.Integer.parseInt(Integer.java:476)
6      at foo.getResult(foo.java:169)
7      at foo.main(foo.java:214)
8
9  $ java bar input.xml
10 Exception in thread "main" java.lang.NullPointerException
11      at bar.printGamblers(bar.java:186)
12      at bar.main(bar.java:52)
13
14 $ java baz input.xml
15 Gambler John Doe lost 0.
16 Gambler Johnny Average lost 0.
17 Gambler Betty Bet lost 0.
18 Gambler Linda Loser lost 0.
19 Gambler Robert Johnson lost 0.
20
21 $ █

```

DTDs—Document Type Definitions

- The XML Recommendation¹² includes technology that enables applications to rigidly specify the XML **dialect** (the **document type**) they expect to see: **DTDs** (Document Type Definitions).
- XML parsers use the DTD to ensure that input data is not only well-formed but also conforms to the DTD (XML speak: input data is **valid**).



Valid XML documents \subset Well-formed XML documents

- **Document validation** is critical, if
 - distinct organizations (B2B) need to share XML data
⇒ also **share the DTDs**,
 - applications need to discover and explore yet unknown XML dialects,
 - high-speed XML throughput is required (once the input is validated, we can **abandon** a lot of **runtime checks**).

¹²<http://www.w3.org/TR/REC-xml>

DTDs—Document Type Definitions

- A document's DTD is directly attached to its XML text using a DOCTYPE declaration:

DOCTYPE Declaration

```

1  <?xml version="1.0"?>
2  <!DOCTYPE t de di>
3  <t>
4  ...
5  </t>
```

- The DOCTYPE declaration follows the XML declaration (`<?xml...?>`) (comments `<!--...-->`, processing instructions `<?...?>` in between are OK).
- The first parameter t of the DOCTYPE declaration is required to match the document's **root element** tag.
- The document type definition itself consists of an **external subset** ($d_e \equiv \text{SYSTEM } \langle uri \rangle$) as well as an **internal subset** ($d_i \equiv [...]$), i.e., embedded in the document itself).
- Both subsets are optional. Should clashes occur, declarations in the internal subset override those in the external subset.

Example:


```
<!DOCTYPE strip
  SYSTEM "file:///DilbertML.dtd" [ <!ENTITY phb "Pointy-Haired Boss"> ] >
```

external subset
internal subset

The ELEMENT Declaration

- The DTD ELEMENT declaration, in some sense, defines the *vocabulary* available in an XML dialect.
- Any XML element *t* to be used in the dialect needs to be introduced via

```
<!ELEMENT t cm>
```

- The **content model** *cm* of the element defines which **element content** is considered valid.
- Whenever an application encounters a *t* element  **anywhere** in a valid document, it may assume that *t*'s content conforms to *cm*.

Content model	Valid content
ANY	arbitrary well-formed XML content
EMPTY	no child elements allowed (attributes OK)
regular expression over tag names, #PCDATA, and constructors , , , +, *, ?	order and occurrence of child elements and text content must <i>match</i> the regular expression

N.B.

- A DTD with `<!ELEMENT t ANY >` gives the application no clue about *t*'s content. Use judiciously.
- A `<!ELEMENT t EMPTY >` forbids any content for *t* elements.

Example: (X)HTML `img`, `br` tags:

XHTML 1.0 Strict DTD

```
1 <!ELEMENT img EMPTY>
2 ...
3 <!ELEMENT br EMPTY>
```

- **Regular expression** content models provide control over the exact order and occurrence of child nodes below an element node:

Reg. exp.	Semantics
<i>t</i> (tag name)	child element with tag <i>t</i>
#PCDATA	text content (parsed character data)
<i>c</i> ₁ , <i>c</i> ₂	<i>c</i> ₁ followed by <i>c</i> ₂
<i>c</i> ₁ <i>c</i> ₂	<i>c</i> ₁ or, alternatively, <i>c</i> ₂
<i>c</i> ⁺	<i>c</i> , one or more times
<i>c</i> [*]	<i>c</i> , zero or more times
<i>c</i> [?]	optional <i>c</i>

Example (DilbertML):

```

DilbertML.dtd
1  <!ELEMENT panel (scene, bubbles*) >
2  <!ELEMENT scene (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble (#PCDATA) >
```

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ ↺ ↻ 🔍

- **Regular expression** content models provide control over the exact order and occurrence of child nodes below an element node:

Reg. exp.	Semantics
<i>t</i> (tag name)	child element with tag <i>t</i>
#PCDATA	text content (parsed character data)
<i>c</i> ₁ , <i>c</i> ₂	<i>c</i> ₁ followed by <i>c</i> ₂
<i>c</i> ₁ <i>c</i> ₂	<i>c</i> ₁ or, alternatively, <i>c</i> ₂
<i>c</i> ⁺	<i>c</i> , one or more times
<i>c</i> [*]	<i>c</i> , zero or more times
<i>c</i> [?]	optional <i>c</i>

Example (DilbertML):

```

DilbertML.dtd
1  <!ELEMENT panel (scene, bubbles*) >
2  <!ELEMENT scene (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble (#PCDATA) >
```

◀ ▶ ⏪ ⏩ ⏴ ⏵ ⏶ ⏷ ⏸ ⏹ ⏺ ⏻ ⏼ ⏽ ⏾ ⏿ ↺ ↻ 🔍

Example (modify bubble element so that we can use `<loud>...</loud>` and `<whisper>...</whisper>` to markup speech more accurately):

```
<bubble>E-mail <loud>two copies</loud> to me when you're done.</bubble>
```

DilbertML.dtd

```
1  <!ELEMENT panel    (scene, bubbles*) >
2  <!ELEMENT scene   (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble  (#PCDATA | loud | whisper)* >
5  <!ELEMENT loud    (#PCDATA) >
6  <!ELEMENT whisper (#PCDATA) >
```

- Element bubble is said to allow **mixed content** (text and element nodes), while panel and bubbles allow **element content** only. Elements scene, loud, whisper have **text content**.
- DTD restriction:
The above example shows the only acceptable placements of #PCDATA in content models.



Example (modify bubble element so that we can use `<loud>...</loud>` and `<whisper>...</whisper>` to markup speech more accurately):

```
<bubble>E-mail <loud>two copies</loud> to me when you're done.</bubble>
```

DilbertML.dtd

```
1  <!ELEMENT panel    (scene, bubbles*) >
2  <!ELEMENT scene   (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble  (#PCDATA | loud | whisper)* >
5  <!ELEMENT loud    (#PCDATA) >
6  <!ELEMENT whisper (#PCDATA) >
```

- Element bubble is said to allow **mixed content** (text and element nodes), while panel and bubbles allow **element content** only. Elements scene, loud, whisper have **text content**.
- DTD restriction:
The above example shows the only acceptable placements of #PCDATA in content models.



Example (modify bubble element so that we can use <loud>...</loud> and <whisper>...</whisper> to markup speech more accurately):

```
<bubble>E-mail <loud>two copies</loud> to me when you're done.</bubble>
```

DilbertML.dtd

```

1  <!ELEMENT panel    (scene, bubbles*) >
2  <!ELEMENT scene   (#PCDATA) >
3  <!ELEMENT bubbles (bubble+) >
4  <!ELEMENT bubble  (#PCDATA | loud | whisper)* >
5  <!ELEMENT loud    (#PCDATA) >
6  <!ELEMENT whisper (#PCDATA) >
```

- Element bubble is said to allow **mixed content** (text and element nodes), while panel and bubbles allow **element content** only. Elements scene, loud, whisper have **text content**.
- DTD restriction:
The above example shows the only acceptable placements of #PCDATA in content models.



Element Content vs. Mixed Content

Element bubbles has *element content*: an XML parser will *not* report white space contained in a bubbles element to its underlying application.

Element bubble has *mixed content*: white space (#PCDATA) is regarded essential and thus reported to the application.

Dilbert.xml

```

1  <bubbles>␣
2  ␣␣<bubble>␣
3  ␣␣␣<loud>No coffee</loud>␣
4  ␣␣no research ...␣
5  ␣␣</bubble>␣
6  </bubbles>
```

SAX events

```

startElement (t="bubbles")
startElement (t="bubble")
characters (buf = "␣␣␣ ", len = 4)
startElement (t="loud")
      ⋮
endElement (t="bubble")
endElement (t="bubbles")
```



Ex.: DTD and valid XML encoding academic titles

Academic.xml

```

1  <?xml version="1.0"?>
2  <!DOCTYPE academic [
3    <!ELEMENT academic (Prof?, (Dr, (rernat|emer|phil)*)?,
4      Firstname, Middlename*, Lastname) >
5    <!ELEMENT Prof      EMPTY >
6    <!ELEMENT Dr        EMPTY >
7    <!ELEMENT rernat    EMPTY >
8    <!ELEMENT emer      EMPTY >
9    <!ELEMENT phil      EMPTY >
10   <!ELEMENT Firstname (#PCDATA) >
11   <!ELEMENT Middlename (#PCDATA) >
12   <!ELEMENT Lastname  (#PCDATA) >
13 ]>
14
15 <academic>
16   <Prof/> <Dr/> <emer/>
17   <Firstname>Don</Firstname>
18   <Middlename>E</Middlename>
19   <Lastname>Knuth</Lastname>
20 </academic>

```



The ATTLIST Declaration

- Using the DTD ATTLIST declaration, validation of XML documents is extended to attributes.
- The ATTLIST declaration associates a list of attribute names a_i with their owning element named t :

ATTLIST Declaration

```

1  <!ATTLIST t
2    a1 τ1 d1
3    ...
4    an τn dn
5  >

```

- The **attribute types** τ_i define which values are valid for attribute a_i .
- The **defaults** d_i indicate if a_i is required or optional (and, if absent, if a default value should be assumed for a_i).
- In XML, the attributes of an element are *unordered*. The ATTLIST declaration prescribes *no order* of attribute usage.



- Via **attribute types**, control over the valid attribute values can be exercised:

Attribute Type τ_i	Semantics
CDATA	character data (no <, but <, ...)
$(v_1 v_2 \dots v_m)$	enumerated literal values
ID	value is document-wide unique identifier for owner element
IDREF	references an element via its ID attribute

Example:

Academic.xml (fragment)

```

1  <!ELEMENT academic (Firstname, Middlename*, Lastname) >
2  <!ATTLIST academic
3    title (Prof|Dr) #REQUIRED
4    type CDATA #IMPLIED >
5  >
6
7  <academic title="Dr" type="rer.nat."> ... </academic>

```



- Via **attribute types**, control over the valid attribute values can be exercised:

Attribute Type τ_i	Semantics
CDATA	character data (no <, but <, ...)
$(v_1 v_2 \dots v_m)$	enumerated literal values
ID	value is document-wide unique identifier for owner element
IDREF	references an element via its ID attribute

Example:

Academic.xml (fragment)

```

1  <!ELEMENT academic (Firstname, Middlename*, Lastname) >
2  <!ATTLIST academic
3    title (Prof|Dr) #REQUIRED
4    type CDATA #IMPLIED >
5  >
6
7  <academic title="Dr" type="rer.nat."> ... </academic>

```



- Attribute defaulting in DTDs:

Attribute Default d_i	Semantics
#REQUIRED	element must have attribute a_i
#IMPLIED	attribute a_i is optional
" v "	attribute a_i is optional , if absent, default value v for a_i is assumed
#FIXED v	attribute a_i is optional , if present, must have value v

Example:

DilbertML.dtd (fragment)

```

1  <!DOCTYPE strip [
2      ...
3      <!ELEMENT characters (character+) >
4      <!ATTLIST characters
5          alphabetical (yes|no) "no" >    <!-- play safe -->
6      <!ELEMENT character (#PCDATA) >
7  ]>

```

- Attribute defaulting in DTDs:

Attribute Default d_i	Semantics
#REQUIRED	element must have attribute a_i
#IMPLIED	attribute a_i is optional
" v "	attribute a_i is optional , if absent, default value v for a_i is assumed
#FIXED v	attribute a_i is optional , if present, must have value v

Example:

DilbertML.dtd (fragment)

```

1  <!DOCTYPE strip [
2      ...
3      <!ELEMENT characters (character+) >
4      <!ATTLIST characters
5          alphabetical (yes|no) "no" >    <!-- play safe -->
6      <!ELEMENT character (#PCDATA) >
7  ]>

```

Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data.
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML element nodes a and b



- attach a unique **identifier** to node b (using an ID attribute), then
- refer** to b from a via this identifier (using an IDREF attribute),
- for an outdegree > 1 (see below), use an IDREFS attribute.



Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data.
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML element nodes a and b



- attach a unique **identifier** to node b (using an ID attribute), then
- refer** to b from a via this identifier (using an IDREF attribute),
- for an outdegree > 1 (see below), use an IDREFS attribute.



Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data.
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML element nodes a and b



- 1 attach a unique **identifier** to node b (using an ID attribute), then
- 2 **refer** to b from a via this identifier (using an IDREF attribute),
- 3 for an outdegree > 1 (see below), use an IDREFS attribute.

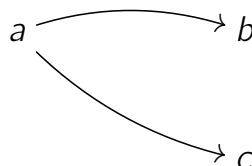


Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data.
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML element nodes a and b



- 1 attach a unique **identifier** to node b (using an ID attribute), then
- 2 **refer** to b from a via this identifier (using an IDREF attribute),
- 3 for an outdegree > 1 (see below), use an IDREFS attribute.

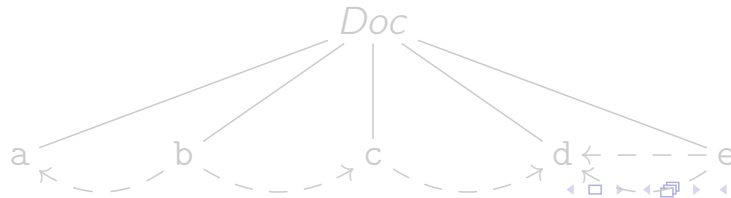


Graph.xml

```

1  <?xml version="1.0"?>
2  <!DOCTYPE graph [
3    <!ELEMENT graph (node+) >
4    <!ELEMENT node ANY >      <!-- attach arbitrary data to a node -->
5    <!ATTLIST node
6      id    ID    #REQUIRED
7      edges IDREFS #IMPLIED > <!-- we may have nodes with outdegree 0 -->
8  ]>
9
10 <graph>
11   <node id="A">a</node>
12   <node id="B" edges="A C">b</node>
13   <node id="C" edges="D">c</node>
14   <node id="D">d</node>
15   <node id="E" edges="D D">e</node>
16 </graph>

```

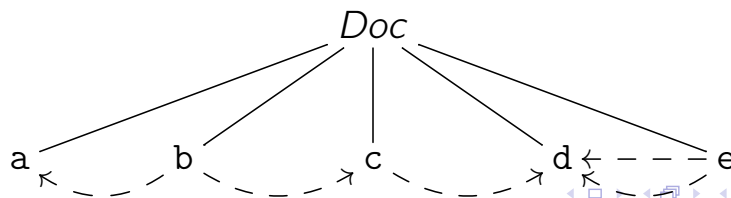


Graph.xml

```

1  <?xml version="1.0"?>
2  <!DOCTYPE graph [
3    <!ELEMENT graph (node+) >
4    <!ELEMENT node ANY >      <!-- attach arbitrary data to a node -->
5    <!ATTLIST node
6      id    ID    #REQUIRED
7      edges IDREFS #IMPLIED > <!-- we may have nodes with outdegree 0 -->
8  ]>
9
10 <graph>
11   <node id="A">a</node>
12   <node id="B" edges="A C">b</node>
13   <node id="C" edges="D">c</node>
14   <node id="D">d</node>
15   <node id="E" edges="D D">e</node>
16 </graph>

```



Example (Character references in DilbertML)

DilbertML.dtd (fragment)

```

1  <!DOCTYPE strip [
2      ...
3      <!ELEMENT character (#PCDATA) >
4      <!ATTLIST character
5          id      ID              #REQUIRED >
6      <!ELEMENT bubble   (#PCDATA) >
7      <!ATTLIST bubble
8          speaker IDREF          #REQUIRED
9          to      IDREFS        #IMPLIED
10         tone    (angry|question|...) #IMPLIED >
11 ]>

```

Validation results (messages generated by Apache's Xerces):

- Setting attribute to to some random non-existent character identifier:
ID attribute 'yoda' was referenced but never declared
- Using a non-enumerated value for attribute tone:
Attribute 'tone' does not match its defined enumeration list



DilbertML.dtd

```

1  <!DOCTYPE strip [
2      <!ELEMENT strip      (prolog, panels) >
3      <!ATTLIST strip
4          copyright CDATA #IMPLIED
5          year      CDATA #IMPLIED >
6      <!ELEMENT prolog    (series, author, characters) >
7
8      <!ELEMENT series    (#PCDATA) >
9      <!ATTLIST series
10         href CDATA #IMPLIED >
11
12     <!ELEMENT author    (#PCDATA) >
13
14     <!ELEMENT characters (character+) >
15     <!ATTLIST characters
16         alphabetical (yes|no) 'no' >
17
18     <!ELEMENT character  (#PCDATA) >
19     <!ATTLIST character
20         id ID #REQUIRED >
21
22     <!ELEMENT panels    (panel+) >
23     <!ATTLIST panels
24         length CDATA #IMPLIED >
25
26     <!ELEMENT panel     (scene, bubbles*) >
27     <!ATTLIST panel
28         no CDATA #IMPLIED >
29
30     <!ELEMENT scene     (#PCDATA) >
31     <!ATTLIST scene
32         visible IDREFS #IMPLIED >
33
34     <!ELEMENT bubbles   (bubble+) >
35
36     <!ELEMENT bubble    (#PCDATA) >
37     <!ATTLIST bubble
38         speaker IDREF #REQUIRED
39         to      IDREFS #IMPLIED
40         tone    (question|angry|screaming) #IMPLIED >
41 ]>

```



Other DTD features

- **User-defined entities** via `<!ENTITY e d>` declarations (usage: `&e;`)

```
<!ENTITY phb "The Pointy-Haired Boss">
```

- **Parameter entities** (“DTD macros”) via `<!ENTITY % e d>` (usage: `%e;`)

```
<!ENTITY ident "ID #REQUIRED">
```

```
...
```

```
<!ATTLIST character
  id %ident; >
```

- **Conditional sections** in DTDs via `<![INCLUDE[...]]>` and `<![IGNORE[...]]>`

```
<!ENTITY % withCharacterIDs "INCLUDE" >
```

```
<!ATTLIST bubble
```

```
<![%withCharacterIDs;[
```

```
  speaker %ident;
```

```
  to      %ident;
```

```
]]>
```

```
tone      (angry|question|...) #IMPLIED >
```



A “Real Life” DTD—GraphML

- GraphML¹³ has been designed to provide a powerful and easy-to-use file format to represent arbitrary graphs.

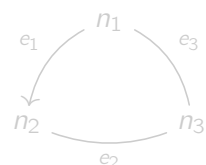
- 1 Graphs (element `graph`) are specified as lists of nodes and edges. Edges point from `source` to `target`.
- 2 Nodes and edges may be annotated using arbitrary descriptions and data.
- 3 Edges may be directed (and attribute `edgedefault` of `graph`).
- 4 Edges may be attached to nodes at specific ports (north, west, ...).

Example:

```

1 <graphml>
2   <graph edgedefault="undirected">
3     <node id="n1"/>
4     <node id="n2"/>
5     <node id="n3"/>
6     <edge id="e1" source="n1" target="n2" directed="true"/>
7     <edge id="e2" source="n2" target="n3" directed="false"/>
8     <edge id="e3" source="n3" target="n1"/>
9   </graph>
10 </graphml>

```



¹³<http://www.graphdrawing.org/>

A “Real Life” DTD—GraphML

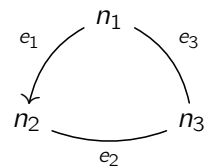
- GraphML¹³ has been designed to provide a powerful and easy-to-use file format to represent arbitrary graphs.
 - 1 Graphs (element graph) are specified as lists of nodes and edges. Edges point from source to target.
 - 2 Nodes and edges may be annotated using arbitrary descriptions and data.
 - 3 Edges may be directed (and attribute edgedefault of graph).
 - 4 Edges may be attached to nodes at specific ports (north, west, ...).

Example:

```

1 <graphml>
2   <graph edgedefault="undirected">
3     <node id="n1"/>
4     <node id="n2"/>
5     <node id="n3"/>
6     <edge id="e1" source="n1" target="n2" directed="true"/>
7     <edge id="e2" source="n2" target="n3" directed="false"/>
8     <edge id="e3" source="n3" target="n1"/>
9   </graph>
10 </graphml>

```



¹³<http://www.graphdrawing.org/>

```

1 <!DOCTYPE graphml [
2
3 <!ELEMENT graphml ((desc)?,(key)*,((data)|(graph))*>
4
5
6 <!ELEMENT locator EMPTY>
7 <!ATTLIST locator
8     xmlns:xlink CDATA #FIXED
9     "http://www.w3.org/TR/2000/PR-xlink-20001220/"
10    xlink:href CDATA #REQUIRED
11    xlink:type (simple) #FIXED "simple"
12 >
13
14 <!ELEMENT desc (#PCDATA)>
15
16
17 <!ELEMENT graph ((desc)?,(((data)|(node)|
18     (edge)|(hyperedge))*|(locator)))>
19 <!ATTLIST graph
20     id ID #IMPLIED
21     edgedefault (directed|undirected) #REQUIRED
22 >
23

```



```

24 <!ELEMENT node      (desc?,(((data|port)*,graph?)|locator))>
25 <!ATTLIST node
26     id          ID          #REQUIRED
27 >
28
29 <!ELEMENT port      ((desc?),((data)|(port))*>
30 <!ATTLIST port
31     name        NMTOKEN    #REQUIRED
32 >
33
34
35 <!ELEMENT edge      ((desc?), (data)*, (graph)?)>
36 <!ATTLIST edge
37     id          ID          #IMPLIED
38     source      IDREF       #REQUIRED
39     sourceport  NMTOKEN     #IMPLIED
40     target      IDREF       #REQUIRED
41     targetport  NMTOKEN     #IMPLIED
42     directed    (true|false) #IMPLIED
43 >
44
45
46 <!ELEMENT hyperedge ((desc?),((data)|(endpoint))*,(graph)?)>

```



```

47 <!ATTLIST hyperedge
48     id          ID          #IMPLIED
49 >
50
51 <!ELEMENT endpoint  ((desc)?)>
52 <!ATTLIST endpoint
53     id          ID          #IMPLIED
54     node        IDREF       #REQUIRED
55     port        NMTOKEN     #IMPLIED
56     type        (in|out|undir) "undir"
57 >
58
59
60 <!ELEMENT key        (#PCDATA)>
61 <!ATTLIST key
62     id          ID          #REQUIRED
63     for         (graph|node|edge|hyperedge|port|endpoint|all) "all"
64 >
65
66 <!ELEMENT data       (#PCDATA)>
67 <!ATTLIST data
68     key         IDREF       #REQUIRED
69     id          ID          #IMPLIED

```



```
70 >  
71 |  
72 ]>
```

Concluding remarks

- DTD syntax:
 - Pro:** compact, easy to understand
 - Con:** not in XML
- DTD functionality:
 - no distinguishable types (everything is character data)
 - no further value constraints (*e.g.*, cardinality of sequences)
 - no support for XML name spaces
- **From a database perspective, DTDs are a poor schema definition language.**

(But: see XML Schema below...)

XML Schema

- With **XML Schema**¹⁴, the W3C provides a **schema description language** for XML documents that goes way beyond the capabilities of the “native” DTD concept.

Specifically:

- ① XML Schema **descriptions are valid XML documents** themselves.
- ② XML Schema provides a **rich set of built-in data types**.
(Modelled after the SQL and Java type systems.)
- ③ **Far-reaching control over the values** a data type can assume (**facets**).
- ④ Users can extend this type system via **user-defined types**.
- ⑤ XML element (and attribute) types may even be derived by **inheritance**.

XML Schema vs. DTDs

Ad ①: Why would you consider this an advantage?

Ad ②: What are the data types supported by DTDs?

¹⁴<http://www.w3.org/TR/xmlschema-0/>

XML Schema

- With **XML Schema**¹⁴, the W3C provides a **schema description language** for XML documents that goes way beyond the capabilities of the “native” DTD concept.

Specifically:

- ① XML Schema **descriptions are valid XML documents** themselves.
- ② XML Schema provides a **rich set of built-in data types**.
(Modelled after the SQL and Java type systems.)
- ③ **Far-reaching control over the values** a data type can assume (**facets**).
- ④ Users can extend this type system via **user-defined types**.
- ⑤ XML element (and attribute) types may even be derived by **inheritance**.

XML Schema vs. DTDs

Ad ①: Why would you consider this an advantage?

Ad ②: What are the data types supported by DTDs?

¹⁴<http://www.w3.org/TR/xmlschema-0/>

Some XML Schema Constructs

Declaring an element

- ```
<xsd:element name="author"/>
```

No further typing specified: the author element may contain string values only.

### Declaring an element with bounded occurrence

- ```
<xsd:element name="item" minOccurs="0" maxOccurs="unbounded"/>
```

Absence of minOccurs/maxOccurs implies *exactly once*.

Declaring a typed element

- ```
<xsd:element name="year" type="xsd:date"/>
```

Content of year takes the format YYYY-MM-DD. Other **simple types**: string, boolean, number, float, duration, time, base64Binary, AnyURI, ...

- **Simple types** are considered **atomic** with respect to XML Schema (e.g., the YYYY part of an xsd:date value has to be extracted by the XML application itself).



- Non-atomic **complex types** are built from simple types using **type constructors**.

### Declaring sequenced content

```

1 <xsd:complexType name="Characters">
2 <xsd:sequence>
3 <xsd:element name="character" minOccurs="1" maxOccurs="unbounded"/>
4 </xsd:sequence>
5 </xsd:complexType>
6 <xsd:complexType name="Prolog">
7 <xsd:sequence>
8 <xsd:element name="series"/>
9 <xsd:element name="author"/>
10 <xsd:element name="characters" type="Characters"/>
11 </xsd:sequence>
12 </xsd:complexType>
13 <xsd:element name="prolog" type="Prolog"/>

```

An xsd:complexType may be used anonymously (no name attribute).

- With attribute mixed="true", an xsd:complexType admits **mixed content**.



- New complex types may be **derived** from an existing (base) type.

#### Deriving a new complex type

```

1 <xsd:element name="newprolog">
2 <xsd:complexType>
3 <xsd:complexContent>
4 <xsd:extension base="Prolog">
5 <xsd:element name="colored" type="xsd:boolean"/>
6 </xsd:extension>
7 </xsd:complexContent>
8 </xsd:complexType>
9 </xsd:element>

```

- **Attributes** are declared within their owner element.

#### Declaring attributes

```

1 <xsd:element name="strip">
2 <xsd:attribute name="copyright"/>
3 <xsd:attribute name="year" type="xsd:gYear"/> ...
4 </xsd:element>

```

Other xsd:attribute modifiers: use (required, optional, prohibited), fixed, default.



- The validation of an XML document against an XML Schema declaration goes as far as peeking into the **lexical representation** of simple typed values.

#### Restricting the value space of a simple type (enumeration)

```

1 <xsd:simpleType name="Tone">
2 <xsd:restriction base="xsd:string">
3 <xsd:enumeration value="question"/>
4 <xsd:enumeration value="angry"/>
5 <xsd:enumeration value="screaming"/>
6 </xsd:restriction>
7 </xsd:simpleType>

```

#### Restricting the value space of a simple type (regular expression)

```

1 <xsd:simpleType name="AreaCode">
2 <xsd:restriction base="xsd:string">
3 <xsd:pattern value="0[0-9]+"/>
4 <xsd:minLength value="3"/>
5 <xsd:maxLength value="5"/>
6 </xsd:restriction>
7 </xsd:simpleType>

```

- Other **facets**: length, maxInclusive, minExclusive, ...



## Other XML Schema Concepts

- **Fixed** and **default** element content,
- support for **null values**,
- uniqueness constraints, arbitrary **keys** (specified via XPath), local keys, key references, and referential integrity,
- ...

## Validating XML Documents Against DTDs

- To validate against this DTD ...

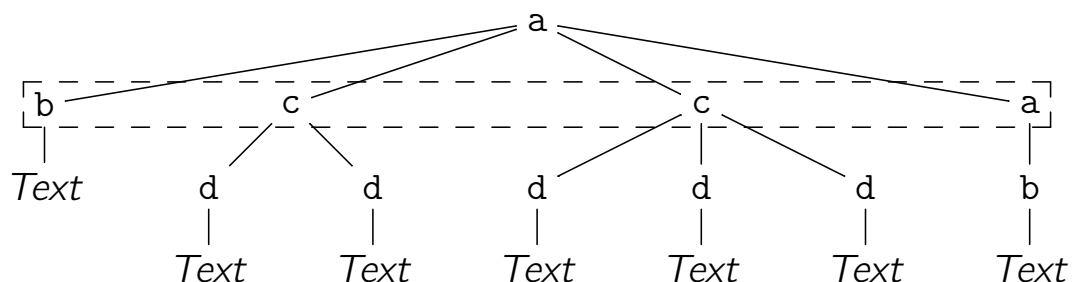
### DTD featuring regular expression (RE) content models

```

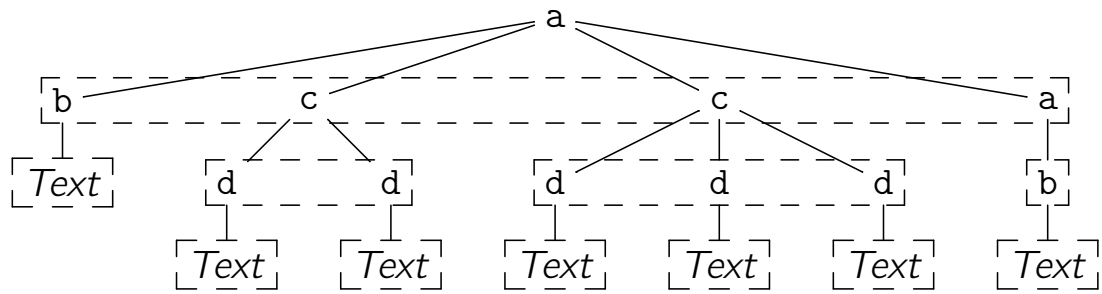
1 <!DOCTYPE a [
2 <!ELEMENT a (b, c*, a?)>
3 <!ELEMENT b (#PCDATA) >
4 <!ELEMENT c (d, d+) >
5 <!ELEMENT d (#PCDATA) >
6]>

```

... means to check that the **sequence of child nodes** for each element **matches** its RE content model:



- When, during RE matching, we encounter a child element  $t$ , we need to **recursively check  $t$ 's content model  $cm(t)$**  in the same fashion:



$$cm(a) = b, c^*, a?$$

$$cm(b) = \#PCDATA$$

$$cm(c) = d, d^+$$

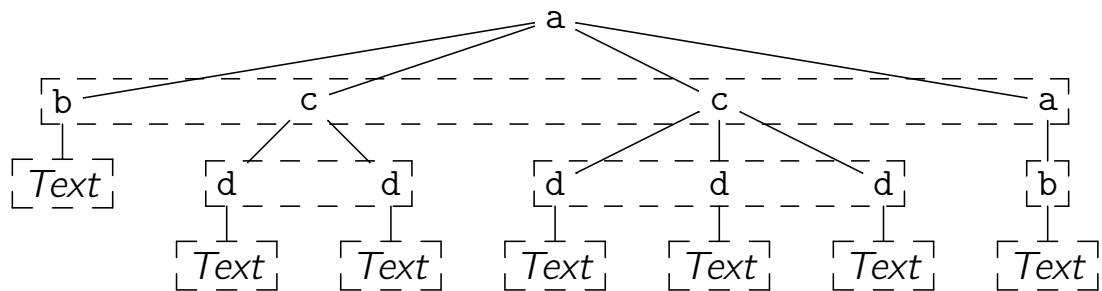
$$cm(d) = \#PCDATA$$

### SAX and DTD validation?

- Can we use SAX to drive this validation (= RE matching) process?
- If so, which SAX events do we need to catch to implement this?



- When, during RE matching, we encounter a child element  $t$ , we need to **recursively check  $t$ 's content model  $cm(t)$**  in the same fashion:



$$cm(a) = b, c^*, a?$$

$$cm(b) = \#PCDATA$$

$$cm(c) = d, d^+$$

$$cm(d) = \#PCDATA$$

### SAX and DTD validation?


- Can we use SAX to drive this validation (= RE matching) process?
- If so, which SAX events do we need to catch to implement this?



## Regular Expressions

- To provide adequate support for SAX-based XML validation, we assume REs of the following structure:

|                  |                                             |
|------------------|---------------------------------------------|
| $RE = \emptyset$ | matches <b>nothing</b>                      |
| $\varepsilon$    | matches <b>empty</b> sequence of SAX events |
| $\#PCDATA$       | matches <i>characters</i> ( $\cdot$ )       |
| $t$              | matches <i>startElement</i> ( $t, \cdot$ )  |
| $RE, RE$         | <b>concatenation</b>                        |
| $RE^+$           | <b>one-or-more repetitions</b>              |
| $RE^*$           | <b>zero-or-more repetitions</b>             |
| $RE?$            | <b>option</b>                               |
| $RE \mid RE$     | <b>alternative</b>                          |
| $(RE)$           |                                             |

-   $\emptyset$  and  $\varepsilon$  are *not* the same thing.
- In the *startElement*( $t, \cdot$ ) callback we can process `<!ATTLIST t ...>` declarations (not discussed here).



- Associated with each RE is the **regular language**  $L(RE)$  (here: sets of sequences of SAX events) this RE **accepts**:

$$\begin{aligned}
 L(\emptyset) &= \emptyset \\
 L(\varepsilon) &= \{\varepsilon\} \\
 L(\#PCDATA) &= \{\textit{characters}(\cdot)\} \\
 L(t) &= \{\textit{startElement}(t, \cdot)\}^{15} \\
 L(RE_1, RE_2) &= \{s_1 s_2 \mid s_1 \in L(RE_1), s_2 \in L(RE_2)\} \\
 L(RE^+) &= \bigcup_{i=1}^{\infty} L(RE^i) \\
 L(RE^*) &= \bigcup_{i=0}^{\infty} L(RE^i) \\
 L(RE?) &= \{\varepsilon\} \cup L(RE) \\
 L(RE_1 \mid RE_2) &= L(RE_1) \cup L(RE_2)
 \end{aligned}$$

- N.B.:**  $RE^0 = \varepsilon$  and  $RE^i = RE, RE^{i-1}$ .

<sup>15</sup>To save trees, we will abbreviate this as  $\{t\}$  from now on.



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 L(\#PCDATA | b^*) & \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 L(\#PCDATA | b^*) & \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 &L(\#PCDATA | b^*) \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 L(\#PCDATA | b^*) & \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$


  $L(d, d^+) = ?$



## Example

- Which sequence of SAX events is matched by the RE `#PCDATA | b*`?

$$\begin{aligned}
 L(\#PCDATA | b^*) & \\
 &= L(\#PCDATA) \cup L(b^*) \\
 &= L(\#PCDATA) \cup \bigcup_{i=0}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup \bigcup_{i=1}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup \bigcup_{i=2}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(b^0) \cup L(b^1) \cup L(b^2) \cup \bigcup_{i=3}^{\infty} L(b^i) \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup L(b, b^1) \cup \dots \\
 &= L(\#PCDATA) \cup L(\varepsilon) \cup L(b) \cup \{s_1 s_2 \mid s_1 \in L(b), s_2 \in L(b^1)\} \cup \dots \\
 &= \{characters(\cdot), \varepsilon, b, bb, \dots\}
 \end{aligned}$$

  $L(d, d^+) = ?$



## Evaluating Regular Expressions (Matching)

- Now that we are this far, we know that matching a sequence of SAX events  $s$  against the content model of element  $t$  means to carry out the test

$$s \stackrel{?}{\in} L(cm(t)) .$$

- $L(cm(t))$ , however, might be infinite or otherwise too costly to construct inside our DTD validator.
- We thus follow a different path that avoids to enumerate  $L(cm(t))$  at all.
- Instead, we will use the **derivative**  $s \setminus RE$  of  $RE$  with respect to input event  $s$ :

$$L(s \setminus RE) = \{s' \mid s s' \in L(RE)\}$$

" $s \setminus RE$  matches everything matched by  $RE$ , with head  $s$  cut off."



## Evaluating Regular Expressions (Matching)

- Now that we are this far, we know that matching a sequence of SAX events  $s$  against the content model of element  $t$  means to carry out the test

$$s \stackrel{?}{\in} L(cm(t)) .$$

- $L(cm(t))$ , however, might be infinite or otherwise too costly to construct inside our DTD validator.
- We thus follow a different path that avoids to enumerate  $L(cm(t))$  at all.
- Instead, we will use the **derivative**  $s \setminus RE$  of  $RE$  with respect to input event  $s$ :

$$L(s \setminus RE) = \{s' \mid s s' \in L(RE)\}$$

" $s \setminus RE$  matches everything matched by  $RE$ , with head  $s$  cut off."



## Evaluating Regular Expressions (Matching)

- Now that we are this far, we know that matching a sequence of SAX events  $s$  against the content model of element  $t$  means to carry out the test

$$s \stackrel{?}{\in} L(cm(t)) .$$

- $L(cm(t))$ , however, might be infinite or otherwise too costly to construct inside our DTD validator.
- We thus follow a different path that avoids to enumerate  $L(cm(t))$  at all.
- Instead, we will use the **derivative**  $s \setminus RE$  of  $RE$  with respect to input event  $s$ :

$$L(s \setminus RE) = \{s' \mid s s' \in L(RE)\}$$

“ $s \setminus RE$  matches everything matched by  $RE$ , with head  $s$  cut off.”



## Evaluating Regular Expressions (Matching)

- Now that we are this far, we know that matching a sequence of SAX events  $s$  against the content model of element  $t$  means to carry out the test

$$s \stackrel{?}{\in} L(cm(t)) .$$

- $L(cm(t))$ , however, might be infinite or otherwise too costly to construct inside our DTD validator.
- We thus follow a different path that avoids to enumerate  $L(cm(t))$  at all.
- Instead, we will use the **derivative**  $s \setminus RE$  of  $RE$  with respect to input event  $s$ :

$$L(s \setminus RE) = \{s' \mid s s' \in L(RE)\}$$

“ $s \setminus RE$  matches everything matched by  $RE$ , with head  $s$  cut off.”



- We can use the derivate operator  $\setminus$  to develop a simple **RE matching procedure**.

Suppose we are to match the SAX event sequence  $s_1s_2s_3$  against  $RE$ :

$$\begin{aligned}
 s_1s_2s_3 \in L(RE) &\Leftrightarrow s_1s_2s_3\varepsilon \in L(RE) \\
 &\Leftrightarrow s_2s_3\varepsilon \in L(s_1 \setminus RE) \\
 &\Leftrightarrow s_3\varepsilon \in L(s_2 \setminus (s_1 \setminus RE)) \\
 &\Leftrightarrow \varepsilon \in L(s_3 \setminus (s_2 \setminus (s_1 \setminus RE))) \ .
 \end{aligned}$$

- We thus have solved our matching problem if
  - 1 we can efficiently **test for  $\varepsilon$ -containment** for a given RE, and
  - 2 we are able to **compute**  $L(s \setminus RE)$  for any given input event  $s$  and any RE.

- We can use the derivate operator  $\setminus$  to develop a simple **RE matching procedure**.

Suppose we are to match the SAX event sequence  $s_1s_2s_3$  against  $RE$ :

$$\begin{aligned}
 s_1s_2s_3 \in L(RE) &\Leftrightarrow s_1s_2s_3\varepsilon \in L(RE) \\
 &\Leftrightarrow s_2s_3\varepsilon \in L(s_1 \setminus RE) \\
 &\Leftrightarrow s_3\varepsilon \in L(s_2 \setminus (s_1 \setminus RE)) \\
 &\Leftrightarrow \varepsilon \in L(s_3 \setminus (s_2 \setminus (s_1 \setminus RE))) \ .
 \end{aligned}$$

- We thus have solved our matching problem if
  - 1 we can efficiently **test for  $\varepsilon$ -containment** for a given RE, and
  - 2 we are able to **compute**  $L(s \setminus RE)$  for any given input event  $s$  and any RE.



 Ad ①: Testing for  $\varepsilon$ 's presence in a regular language.

Define a predicate (Boolean function)  $nullable(RE)$  such that

$$nullable(RE) \Leftrightarrow \varepsilon \in L(RE) .$$

$$nullable(\emptyset) = false$$

$$nullable(\varepsilon) = true$$

$$nullable(\#PCDATA) = false$$

$$nullable(t) =$$

$$nullable(RE_1, RE_2) =$$

$$nullable(RE^+) =$$

$$nullable(RE^*) =$$

$$nullable(RE?) =$$

$$nullable(RE_1 | RE_2) =$$



## Example

Does  $L(\#PCDATA | b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} nullable(\#PCDATA | b^*) &= nullable(\#PCDATA) \vee nullable(b^*) \\ &= false \vee true \\ &= true . \end{aligned}$$

  $nullable(\text{Prof?}, \text{Dr}, (\text{rernat} | \text{emer} | \text{phil})^+) = ?$



## Example

Does  $L(\#PCDATA | b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} \text{nullable}(\#PCDATA | b^*) &= \text{nullable}(\#PCDATA) \vee \text{nullable}(b^*) \\ &= \text{false} \vee \text{true} \\ &= \text{true} . \end{aligned}$$

  $\text{nullable}(\text{Prof?}, \text{Dr}, (\text{rernat} | \text{emer} | \text{phil})^+) = ?$



## Example

Does  $L(\#PCDATA | b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} \text{nullable}(\#PCDATA | b^*) &= \text{nullable}(\#PCDATA) \vee \text{nullable}(b^*) \\ &= \text{false} \vee \text{true} \\ &= \text{true} . \end{aligned}$$

  $\text{nullable}(\text{Prof?}, \text{Dr}, (\text{rernat} | \text{emer} | \text{phil})^+) = ?$



## Example

Does  $L(\#PCDATA \mid b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} \text{nullable}(\#PCDATA \mid b^*) &= \text{nullable}(\#PCDATA) \vee \text{nullable}(b^*) \\ &= \text{false} \vee \text{true} \\ &= \text{true} . \end{aligned}$$

  $\text{nullable}(\text{Prof?}, \text{Dr}, (\text{rernat} \mid \text{emer} \mid \text{phil})^+) = ?$



## Example

Does  $L(\#PCDATA \mid b^*)$  contain the empty SAX event sequence  $\varepsilon$ ?

$$\begin{aligned} \text{nullable}(\#PCDATA \mid b^*) &= \text{nullable}(\#PCDATA) \vee \text{nullable}(b^*) \\ &= \text{false} \vee \text{true} \\ &= \text{true} . \end{aligned}$$

  $\text{nullable}(\text{Prof?}, \text{Dr}, (\text{rernat} \mid \text{emer} \mid \text{phil})^+) = ?$



Ad ②: Note that the **derivative**  $s\backslash$  is an operator on REs (to REs). We define it like follows and justify this definition on the next slides.

$$\begin{aligned}
 s\backslash\emptyset &= \emptyset \\
 s\backslash\varepsilon &= \emptyset \\
 s\backslash\#PCDATA &= \begin{cases} \varepsilon & \text{if } s = \text{characters}(\cdot) \\ \emptyset & \text{otherwise} \end{cases} \\
 s\backslash t &= \begin{cases} \varepsilon & \text{if } s = \text{startElement}(t, \cdot) \quad // \star \text{ recursively match } cm(t) \\ \emptyset & \text{otherwise} \end{cases} \\
 s\backslash(RE_1, RE_2) &= \begin{cases} ((s\backslash RE_1), RE_2) \mid (s\backslash RE_2) & \text{if } nullable(RE_1) \\ (s\backslash RE_1), RE_2 & \text{otherwise} \end{cases} \\
 s\backslash RE^+ &= (s\backslash RE), RE^* \\
 s\backslash RE^* &= (s\backslash RE), RE^* \\
 s\backslash RE? &= s\backslash RE \\
 s\backslash(RE_1 \mid RE_2) &= (s\backslash RE_1) \mid (s\backslash RE_2)
 \end{aligned}$$

## Correctness: Case Analysis I

To assess the correctness of this derivative construction  $s\backslash RE = RE'$  we can systematically check all 9 cases for **language equivalence**, i.e.

$$L(s\backslash RE) = L(RE') .$$

①  $RE = \emptyset$ :

$$\begin{aligned}
 L(s\backslash\emptyset) &= \{s' \mid s s' \in L(\emptyset)\} \\
 &= \{s' \mid s s' \in \emptyset\} \\
 &= \emptyset \\
 &= L(\emptyset).
 \end{aligned}$$

## Correctness: Case Analysis II

②  $RE = \varepsilon$ :

$$\begin{aligned} L(s \setminus \varepsilon) &= \{s' \mid s s' \in L(\varepsilon)\} \\ &= \{s' \mid s s' \in \{\varepsilon\}\} \\ &= \emptyset \\ &= L(\emptyset). \end{aligned}$$

③  $RE = \#PCDATA, s = characters(\cdot)$ :

$$\begin{aligned} L(characters(\cdot) \setminus \#PCDATA) &= \{s' \mid characters(\cdot) s' \in L(\#PCDATA)\} \\ &= \{s' \mid characters(\cdot) s' \in \{characters(\cdot)\}\} \\ &= \{\varepsilon\} \\ &= L(\varepsilon). \end{aligned}$$

## Correctness: Case Analysis III

$RE = \#PCDATA, s \neq characters(\cdot)$ :

$$\begin{aligned} L(s \setminus \#PCDATA) &= \{s' \mid s s' \in L(\#PCDATA)\} \\ &= \{s' \mid s s' \in \{characters(\cdot)\}\} \\ &= \emptyset \\ &= L(\emptyset). \end{aligned}$$

④  $RE = t$ . Analogous to ③.

⑤  $RE = RE_1, RE_2, nullable(RE_1) = false$ :

$$\begin{aligned} L(s \setminus (RE_1, RE_2)) &= \{s' \mid s s' \in L(RE_1, RE_2)\} \\ &= \{s' \mid s' \in L((s \setminus RE_1), RE_2)\} \\ &= L((s \setminus RE_1), RE_2). \end{aligned}$$

## Correctness: Case Analysis IV

$RE = RE_1, RE_2$ ,  $nullable(RE_1) = true$ :

$$\begin{aligned}
 L(s \setminus (RE_1, RE_2)) &= \{s' \mid s s' \in L(RE_1, RE_2)\} \\
 &= \{s' \mid s s' \in L(RE_2) \vee s s' \in L(RE_1, RE_2)\} \\
 &= \{s' \mid s' \in L(s \setminus RE_2) \vee s' \in L((s \setminus RE_1), RE_2)\} \\
 &= \{s' \mid s' \in L(s \setminus RE_2)\} \cup \{s' \mid s' \in L((s \setminus RE_1), RE_2)\} \\
 &= L(s \setminus RE_2) \cup L((s \setminus RE_1), RE_2) \\
 &= L((s \setminus RE_2) \mid ((s \setminus RE_1), RE_2)).
 \end{aligned}$$

## Correctness: Case Analysis V

⑥  $RE = RE_1 \mid RE_2$ :

$$\begin{aligned}
 L(s \setminus (RE_1 \mid RE_2)) &= \{s' \mid s s' \in L(RE_1 \mid RE_2)\} \\
 &= \{s' \mid s s' \in L(RE_1) \cup L(RE_2)\} \\
 &= \{s' \mid s s' \in L(RE_1)\} \cup \{s' \mid s s' \in L(RE_2)\} \\
 &= \{s' \mid s' \in L(s \setminus RE_1)\} \cup \{s' \mid s' \in L(s \setminus RE_2)\} \\
 &= L(s \setminus RE_1) \cup L(s \setminus RE_2) \\
 &= L((s \setminus RE_1) \mid (s \setminus RE_2)).
 \end{aligned}$$

## Correctness: Case Analysis VI

7  $RE = RE^*$ ,  $nullable(RE) = false$ :

$$\begin{aligned}
 L(s \setminus RE^*) &= L(s \setminus (\varepsilon \mid (RE, RE^*))) \\
 &= L(s \setminus \varepsilon) \cup L(s \setminus (RE, RE^*)) \\
 &= L(s \setminus (RE, RE^*)) \\
 &= L((s \setminus RE), RE^*).
 \end{aligned}$$

$RE = RE^*$ ,  $nullable(RE) = true$ :

$$\begin{aligned}
 L(s \setminus RE^*) &= L(s \setminus (\varepsilon \mid (RE, RE^*))) \\
 &= L((s \setminus \varepsilon) \mid (s \setminus (RE, RE^*))) \\
 &= L(\emptyset \mid (s \setminus (RE, RE^*))) \\
 &= L(s \setminus (RE, RE^*)) \\
 &= L((s \setminus RE^*) \mid ((s \setminus RE), RE^*)) \\
 &= L(s \setminus RE^*) \cup L((s \setminus RE), RE^*) \\
 &= L((s \setminus RE), RE^*).
 \end{aligned}$$

## Correctness: Case Analysis VII

8  $RE = RE^+$ . Follows from  $RE^+ = RE, RE^*$ .

9  $RE = RE?$ . Follows from  $RE? = \varepsilon \mid RE$ .

## Matching SAX events against an RE

RE content model  $b, c^*, a?$  is to be matched against SAX events  $bcca$ .<sup>16</sup>

To validate,

- ① construct the corresponding derivative  $RE' = a \setminus (c \setminus (c \setminus (b \setminus (b, c^*, a?))))$ ,
- ② then test  $nullable(RE')$ .

**Hint:** To simplify phase ①, use the following **laws**, valid for REs in general:

$$\begin{array}{ll}
 \varepsilon^* & = \varepsilon & \varepsilon, RE & = RE \\
 \emptyset^* & = \varepsilon & \emptyset, RE & = \emptyset \\
 \varepsilon^+ & = \varepsilon & RE, \varepsilon & = RE \\
 \emptyset^+ & = \emptyset & RE, \emptyset & = \emptyset \\
 \varepsilon? & = \varepsilon & \emptyset | RE & = RE \\
 \emptyset? & = \varepsilon & RE | \emptyset & = RE
 \end{array}$$

<sup>16</sup>Actual event sequence:

$startElement(b, \cdot), startElement(c, \cdot), startElement(c, \cdot), startElement(a, \cdot)$ . 

## Plugging It All Together

The following SAX callbacks use the aforementioned RE matching techniques to (partially) implement DTD validation **while parsing** the input XML document:

The input DTD (declaring the content models  $cm(\cdot)$ ) is

```
<!DOCTYPE r [...]>
```

$startDocument()$

```
 $S.empty();$
 $RE \leftarrow cm(r);$
return;
```

$startElement(t, \cdot)$

```
 $RE \leftarrow t \setminus RE;$
 $S.push(RE);$
 $RE \leftarrow cm(t);$
return;
```

$endElement(t)$


```
if $nullable(RE)$ then
 | $RE \leftarrow S.pop();$
else
 | $\star FAIL \star;$
return;
```

$characters(\cdot)$

```
 $RE \leftarrow \#PCDATA \setminus RE;$
return;
```

$endDocument()$

```
 $\star OK \star;$
```

**N.B.** Stack  $S$  is used to suspend [resume] the RE matching for a specific element node whenever SAX descends [ascends] the XML document tree. 



## Plugging It All Together

The following SAX callbacks use the aforementioned RE matching techniques to (partially) implement DTD validation **while parsing** the input XML document:

The input DTD (declaring the content models  $cm(\cdot)$ ) is

```
<!DOCTYPE r [...]>
```

startDocument()

```
S.empty();
RE ← cm(r);
return;
```

startElement(t, ·)

```
RE ← t\RE;
S.push(RE);
RE ← cm(t);
return;
```

endElement(t)

```
if nullable(RE) then
 RE ← S.pop();
else
 ★ FAIL ★;
return;
```

characters(·)

```
RE ← #PCDATA\RE;
return;
```

endDocument()

```
★ OK ★;
```

**N.B.** Stack  $S$  is used to suspend [resume] the RE matching for a specific element node whenever SAX descends [ascends] the XML document tree.

## Plugging It All Together

The following SAX callbacks use the aforementioned RE matching techniques to (partially) implement DTD validation **while parsing** the input XML document:

The input DTD (declaring the content models  $cm(\cdot)$ ) is

```
<!DOCTYPE r [...]>
```

startDocument()

```
S.empty();
RE ← cm(r);
return;
```

startElement(t, ·)

```
RE ← t\RE;
S.push(RE);
RE ← cm(t);
return;
```

endElement(t)

```
if nullable(RE) then
 RE ← S.pop();
else
 ★ FAIL ★;
return;
```

characters(·)

```
RE ← #PCDATA\RE;
return;
```

endDocument()

```
★ OK ★;
```

**N.B.** Stack  $S$  is used to suspend [resume] the RE matching for a specific element node whenever SAX descends [ascends] the XML document tree.