XML

The acronym XML stands for **eXtensible Markup Language**. It is a language for defining unranked trees with plain text, such that the syntax is easy to learn, write and understand, both for humans and computer programs.

These trees are used to model **structured text documents**.

To understand what XML is and how this modelling works, it is probably easier to start with a small example.

Consider an e-mail. What are the different **elements** and what is the **structure**, that is, how are the elements related to each other?

XML/Example

As far as the elements are concerned, an e-mail contains at least

- the recipient's address,
- a subject,
- the sender's address,
- a body of plain text.

The elements correspond to the tree nodes and the structure is modeled by the shape of the tree itself.

XML/Example (cont)

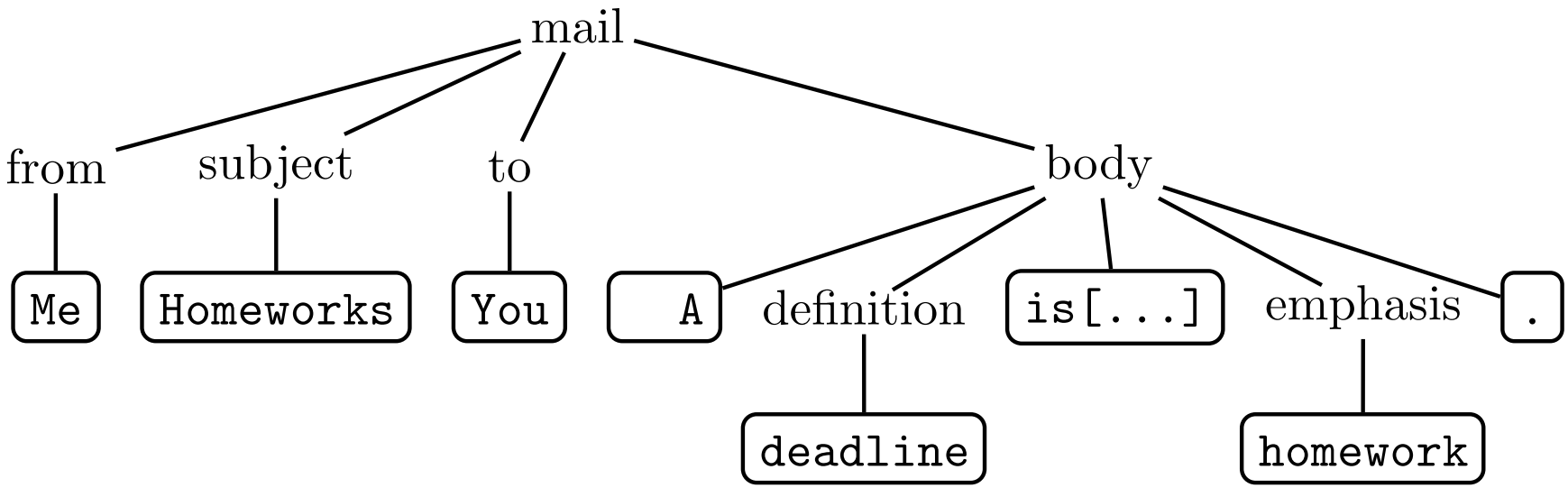
For example:

```
From: Me
Subject: Homeworks
To: You

  A deadline is a due date for your homework.
```

XML/Example (cont)

This e-mail can be modeled by a tree as follows:



Notice that the (boxed) leaves contain text whereas the inner nodes contain information **about** their subtrees, in particular the leaves.

XML/Example (cont)

Since the information of the inner nodes describes the information actually laid out, it is called **meta-data** or **mark-up**. The way to write this document in XML is as follows.

```
<mail>
  <from>Me</from>
  <subject>Homeworks</subject>
  <to>You</to>
  <body>
  A <definition>deadline</definition> is a due date for your
<emphasis>homework</emphasis>.
  </body>
</mail>
```

XML/Tags

Each subtree is denoted by an opening and a closing **tag**. An opening tag is a name enclosed between < and >. A closing tag is a name enclosed between </ and >.

The **tag name** is not part of the text, it is meta-data, so it suggests the meaning of the data contained in the subtree.

For example, the whole XML document is enclosed in tags whose name is "`mail`" because the document describes a mail.

Note the tag names "`body`", "`definition`" and "`emphasis`": this is the way we interpreted the red colour and the italics in the mail, but other interpretations are possible: such interpretations are **not** defined in XML.

XML/Elements, nodes and declaration

An XML tree is called an **element** in XML parlance. In particular, the element including all the others is called the **root element** (here, it is named "`mail`").

A node in the XML tree corresponds, for now, to the opening and closing tags only. The nodes have the same order as the elements.

The data (as opposed to the meta-data) is always contained in the leaves, and is always text.

Our example is not exactly a correct XML document because it lacks a special element which says that the document is indeed XML, and, more precisely, what is the version of XML used here, e.g.,

```
<?xml version="1.0"?>
```

XML/Declaration and empty elements

This special element is actually not an element, as the special markers `<?` and `?>` tend to show. It is more a declaration, some information about the current file, to destination of the reader, whether it is a parsing software, usually called an **XML processor**, or a human.

Consider now the following element:

```
<axiom>
The empty set <empty/> contains no elements.
</axiom>
```

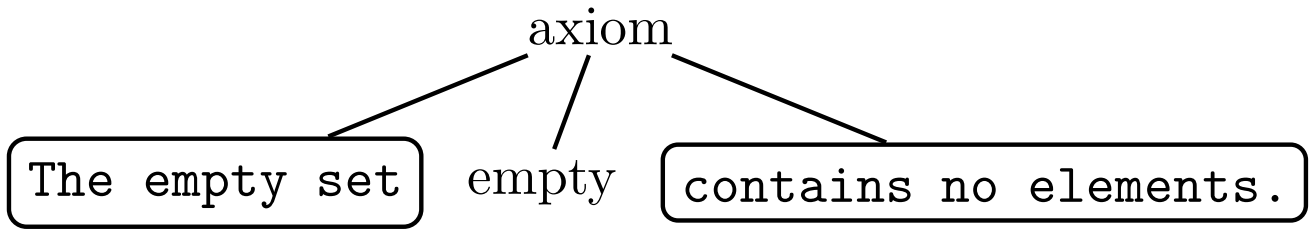
which could be interpreted as

Axiom: The empty set $\varnothing$ contains no elements.

XML/Empty elements (cont)

This `<empty/>` is an **empty element**, it has a special syntax for ending the tag, `/>`, and it is neither an opening nor a closing tag.

It is useful for denoting things, as symbols, that cannot be written as text and need to be distinguished from text.



An empty element corresponds to a leaf in the XML tree, despite it is meta-data data and not data.

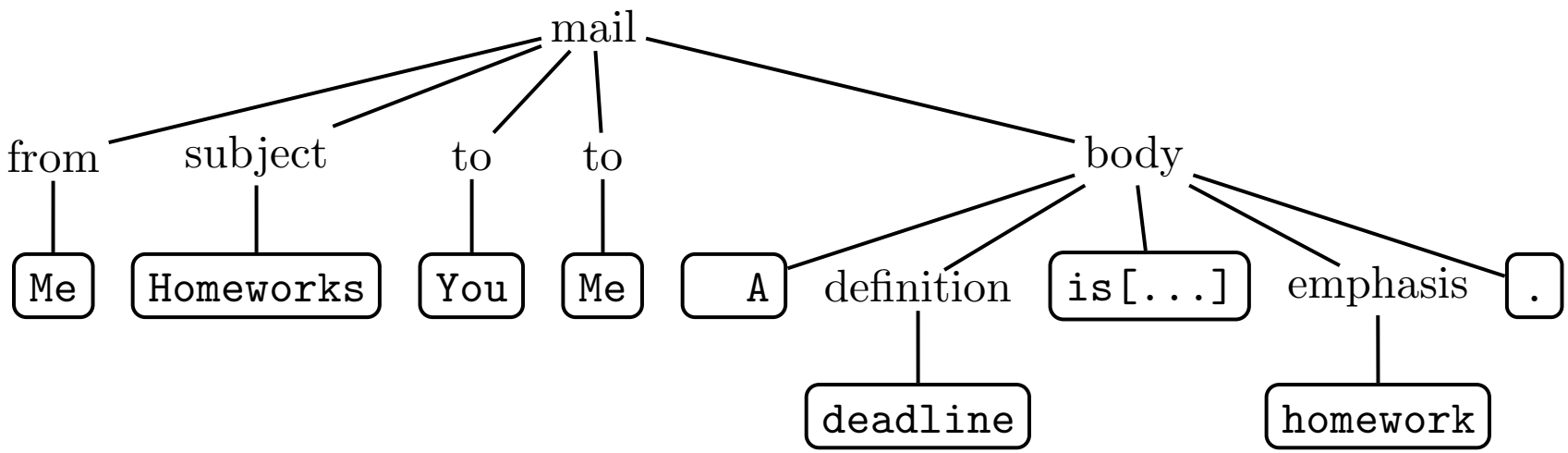
XML/Repeated nodes

Nodes do not need to be unique at a given tree level. For instance, if we want to send a mail to several recipients we would write:

```
<mail>
  <from>Me</from>
  <subject>Homeworks</subject>
  <to>You</to>
  <to>Me</to>
  <body>
  A <definition>deadline</definition> is a due date for your
<emphasis>homework</emphasis>.
  </body>
</mail>
```

XML/Repeated nodes (cont)

The XML tree associated to this XML document is



Note that there are two nodes "to" at the same level, and that their order must be the same as in the XML document.

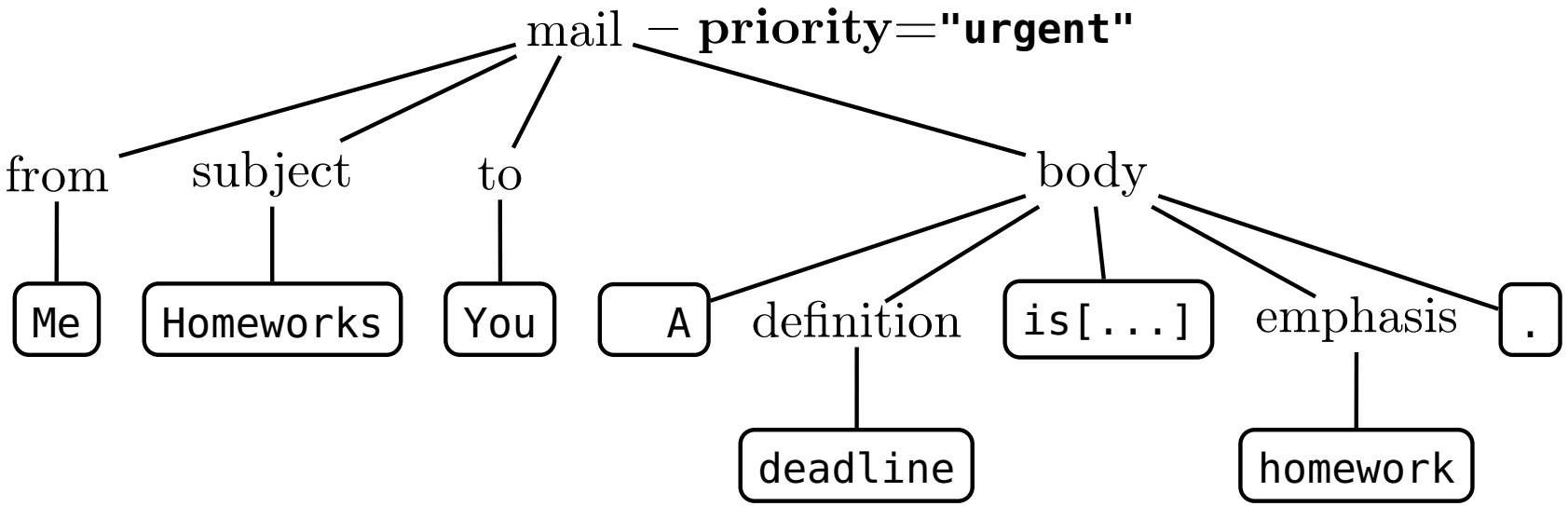
XML/Attributes

It is possible to annotate each meta-data node with some labeled strings, called **attributes**. For example, we may want to specify that our mail is urgent, which is a property of the mail as a whole, not a part of the contents per se:

```
<mail priority="urgent">
  <from>Me</from>
  <subject>Homeworks</subject>
  <to>You</to>
  <body>
  A <definition>deadline</definition> is a due date for your
<emphasis>homework</emphasis>.
  </body>
</mail>
```

XML/Attributes (cont)

It is possible to represent that XML document by the annotated tree



XML/Attributes (cont)

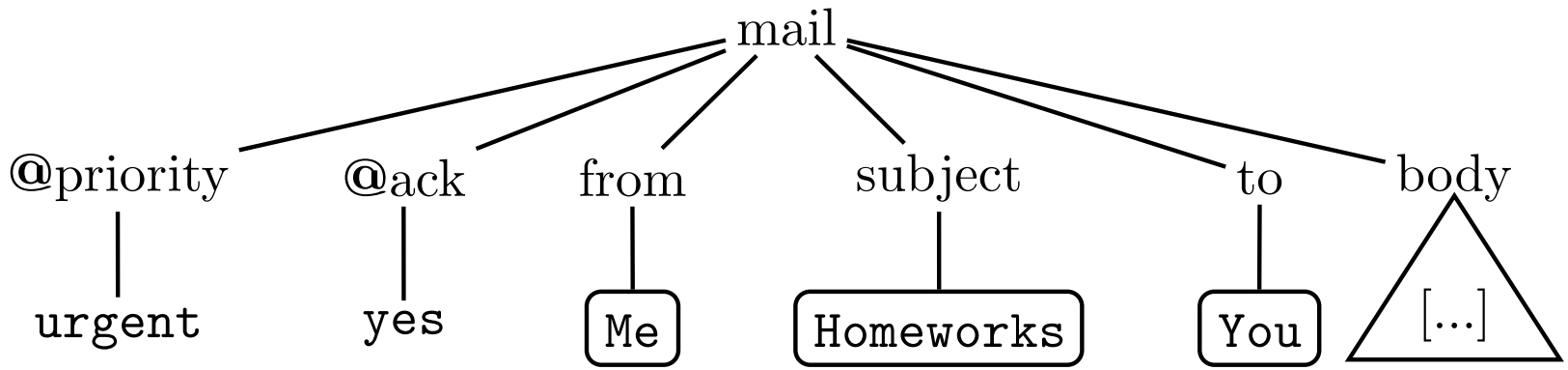
It is possible to attach several attributes to a given element, like

```
<mail priority="urgent" ack="yes">
  <from>Me</from>
  <subject>Homeworks</subject>
  <to>You</to>
  <body>
  A <definition>deadline</definition> is a due date for your
<emphasis>homework</emphasis>.
  </body>
</mail>
```

The order of the attributes matters. Any element can have attributes, including empty elements.

XML/Attributes (cont)

Attributes are considered to be a special kind of node, although they are not often represented as such for room's sake.



Note the symbol @ preceding the attribute name, which distinguishes it from element nodes. At a given tree level, the attribute nodes are placed *before* the element nodes.

XML/Attributes (cont)

The declaration can hold several attributes, besides `version`, like

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml version='1.1' encoding="US-ASCII"?>
<?xml version= '1.0' encoding='iso-8859-1'?>
```

The encoding is the **character encoding** of the XML document, which is particularly useful when using unicode or some Asian fonts.

Note that the attributes must be in lowercase, the *value* of the attributes can be enclosed in single or double quotes.

In the case of `version` and `encoding`, only some standardized values are valid.

XML/Escaping characters

All programming languages offer strings of characters to the programmer to use. For instance, in C, the strings are enclosed between double quotes: `"abc"`.

Thus, if the string contains double-quotes, we must take care of **escaping** them, so the compiler (more precisely: the parser) can distinguish the double-quotes in the contents from the enclosing double-quotes.

In C, character escaping is achieved by adding a backslash just before the character, e.g., `"He said: \"Hello!\"."` is a valid C string.

In XML, there is a similar problem. The attribute values can either be enclosed by single or double quotes. If the latter, the double-quotes in the contents need escaping; if the former, the quotes need escaping. Problems also stem from the characters used for the mark-up.

XML/Escaping characters (cont)

For example, the following element

```
<problem>For all integer n, we have n < n + 1.</problem>
```

is not valid because the text between the tags contains the character "<", which is confused by the XML parsers with the (expected) start of a tag:

```
<problem>For all integer n, we have n < n + 1.</problem>
```

The XML way to escape this character is to use instead the special sequence of characters `<` so the previous, corrected, element becomes

```
<valid>For all integer n, we have n &lt; n + 1.</valid>
```

XML/Predefined named entities

The sequence `<` is called a **predefined named entity**.

Such entities always

1. start with an ampersand (`&`),
2. continue with a predefined name (here, `lt`),
3. end with a semi-colon (`;`).

The choice of the ampersand to mark the start of a predefined named entity entails that this very character must itself be escaped...

So one must always use `&` instead.

XML/Predefined named entities (cont)

There are some other characters which can *sometimes* cause a problem to XML parsers (as opposed to always create a problem, as < and & do).

A summary of all the predefined named entities is given in the following table.

Character	Entity	Mandatory
&	`&`	always
<	`<`	always
>	`>`	in attribute values
"	`&quot;`	in double-quoted strings
'	`&apos;`	in single-quoted strings

XML/Predefined named entities (cont)

Consider

```
<?xml version="1.0" encoding="UTF-8"?>
<escaping>
  <amp>&amp;amp;</amp>
  <lt>&lt;</lt>
  <quot>&quot;</quot>
  <quot attr="&quot;">"</quot>
  <apos attr='&apos;'>&apos;</apos>
  <apos>'</apos>
  <gt>&gt;</gt>
  <gt attr="&gt;">></gt>
  <other>&#100;</other>
  <other>&#x00E7;</other>
</escaping>
```

XML/Predefined numbered entities

The two last entities are **predefined numbered entities** because they denote characters by using their unicode code (which ranges from 0 to 65,536).

Check out `http://www.unicode.org/` for unicode.

If the code is given in decimal (i.e., using base 10), it is introduced by `&#`, e,g, `&#100`.

If the code is given in hexadecimal (i.e., using base 16), it is introduced by `&#x`, e.g., `&#x00E7`.

XML/User-defined internal entities and document type declarations

It can be annoying to use numbers to refer to characters, especially if one considers that unicode requires four digits.

To make life easier, it is possible to bind a name to an entity representing a character: a **user-defined internal entity**.

They are called internal because their definition must be in the same document where they are used.

XML/User-defined internal entities and document type declarations (cont)

For example, it is easier to use `&n;` instead of `&#241`, especially if the text is in Spanish (this represents the letter ñ).

This kind of entity must be declared in the **document type declaration**, which is located, if any, just after the declaration `<?xml ... ?>` and before the root element.

XML/User-defined internal entities and document type declarations (cont)

A document type declaration is made, from left to right, of

1. an opening tag `<!DOCTYPE`,
2. the root element name,
3. the character `[`,
4. the named character entity declarations,
5. the closing tag `]>`

XML/User-defined internal entities and document type declarations (cont)

A named character entity declaration is made, from left to right, of

1. the opening tag `<!ENTITY`,
2. the entity name,
3. the numbered character entity between double-quotes,
4. the closing tag `>`

For example:

```
<!ENTITY n "&#241;">
```

XML/User-defined internal entities and document type declarations (cont)

A complete example:

```
<?xml version="1.0"?>
<!DOCTYPE spain [
  <!ENTITY n "&#241;">
]>
<spain>
Viva Espa&n;a!
</spain>
```

One can think such an entity as being a macro in CPP, the C preprocessor language.

XML/User-defined internal entities and document type declarations (cont)

It is possible to extend user-defined internal entities to denote any character string, not just a single character.

Typically, if one wishes to repeat a piece of text, like a company name or a person's name, a good idea is to give a name to this string and, wherever one want its contents, an entity with the given name is put instead.

The syntax for the declaration is the same, but more characters are put between double-quotes. For example,

```
<!ENTITY univ "Konkuk University">
<!ENTITY motto "<spain>Viva Espa&n;a!</spain>">
<!ENTITY n "&#241;">
```

XML/External entities

Sometimes the XML document needs to include other XML documents and copying the external documents once is not a good strategy, since this avoids keeping track of the evolution of these external documents.

Fortunately, XML allows to specify the inclusion of other XML documents by means of **external entities**. The declaration of these entities is as follows:

1. an opening tag `<!ENTITY`,
2. the entity name,
3. the keyword `SYSTEM`,
4. the full name of the XML file between double-quotes,
5. the closing tag >

XML/External entities (cont)

For example,

```
<?xml version="1.0"?>
<!DOCTYPE longdoc [
  <!ENTITY part1 SYSTEM "p1.xml">
  <!ENTITY part2 SYSTEM "p2.xml">
  <!ENTITY part3 SYSTEM "p3.xml">
]>
<longdoc>
  The included files are:
  &part1;
  &part2;
  &part3;
</longdoc>
```

XML/External entities (cont)

At parsing time, the external entities are fetched and copied into the main XML document, replacing the entity.

Therefore the included parts cannot contain any prolog, i.e., the XML declaration `<?xml ... ?>` and the document type declaration `<!DOCTYPE ... ]>`, if any.

XML processors are required, when reading an external entity, to copy verbatim the content of the referred external document, and then parse it as if it always belonged to the master document (that is, the one which imports the others).

XML/Unparsed entities and notations

Unparsed entities allow to refer to a binary objects, like images, or some text which is not XML, like a program. They are declared by

1. the opening tag `<!ENTITY`,
2. the entity name,
3. the keyword `SYSTEM`,
4. the full name of the non-XML external file between double-quotes,
5. the keyword `NDATA`,
6. a notation (the kind of the file),
7. the closing tag >

XML/Unparsed entities and notations (cont)

Had we used external entities, the included object had been copied in place of the reference and parsed as XML — which it is not. Consider

```
<?xml version="1.0"?>
<!DOCTYPE doc [
  <!NOTATION gif
     SYSTEM "CompuServe Graphics Interchange Format 87a">
  <!ENTITY picture SYSTEM "picture.gif" NDATA gif>
  <!ENTITY me "Christian Rinderknecht">
]>
<doc>
  <para>The following element refers to my picture:</para>
  <graphic image="picture" alt="A picture of &me;"/>
</doc>
```

XML/Unparsed entities and notations (cont)

Notice the notation "`gif`", which is the kind of the unparsed entity. Notations must be defined in the document type declarations as

1. the opening tag `<!NOTATION`,
2. the notation name,
3. the keyword `SYSTEM`,
4. a description of the kind of unparsed entity the notation refers to (it can be a MIME type, an URL, plain English...)
5. the closing tag >

XML/Unparsed entities and notations (cont)

Notice also that unparsed entities

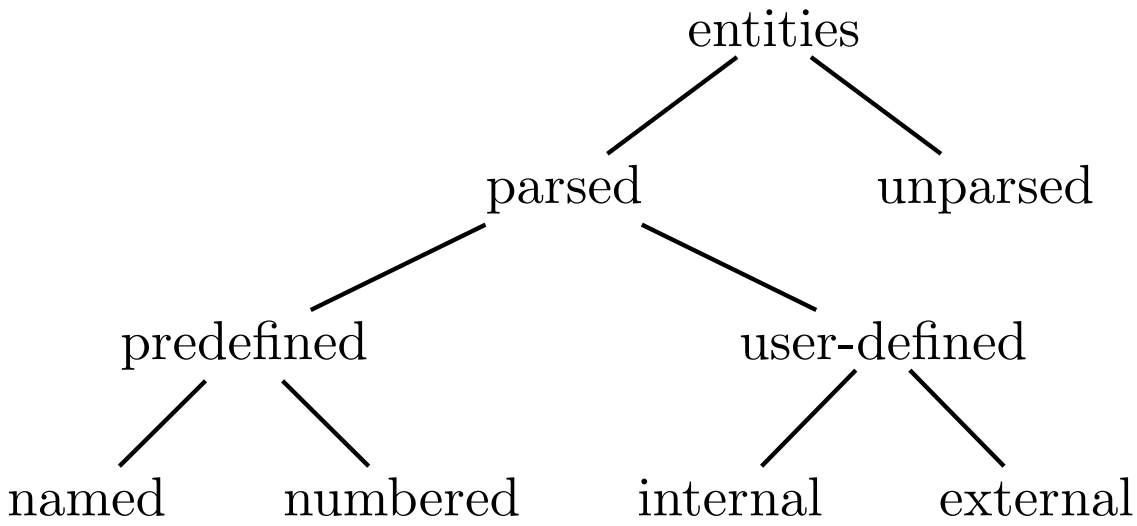
- must be used as attribute values (in our example, the attribute name is "`image`"),
- are names ("`picture`"), instead of the usual entity syntax ("`&picture;`").

XML/Unparsed entities and notations (cont)

This example is **not** well-formed.

```
<?xml version="1.0"?>
<!DOCTYPE doc [
  <!NOTATION jpeg SYSTEM "image/jpeg">
  <!ENTITY pic "pictures/me.jpeg" NDATA jpeg>
]>
<doc>
  &pic;
</doc>
```

XML/A summary of all kinds of entities



XML/Unparsed character data

It is sometimes tiresome to have to escape characters, i.e., use character entities.

To avoid the need of escaping, there is a special construct: **CDATA sections** (short for "Character DATA"), made of

1. an opening tag `<!CDATA[`,
2. some text without escaping and without the sequence `]]>`,
3. a closing tag `]]>`.

For example

```
<para>A test in C:
  <c><![CDATA[if (x < y) return &r;]]></c>
</para>
```

XML/Internal linking

Consider a document representing a technical book, like a textbook. It is common to find cross-references in such kind of books, i.e., references in some chapters to other chapters or sections, or bibliographical entries.

One easy way to achieve this is to use some attributes as labels and some attributes as references.

The problem is that the writer is then in charge of checking whether

- a given label is unique in the whole document,
- every reference is linked to a label.

XML/Internal linking (cont)

XML provides a way to ensure that any validating parser will check this kind of internal linking automatically: using `ID` and `IDREF`.

The former is the kind of all the (attribute) labels and the latter is the kind of all the (attribute) references.

The attributes used either as label or reference must be declared in the `DOCTYPE` section using `ATTLIST`.

XML/Internal linking/Labels

For the labels, use

1. an opening tag `<!ATTLIST`,
2. the name of the element being labelled,
3. the names of the label attributes separated by spaces,
4. the keyword `ID`,
5. the keyword `#REQUIRED` if the element must always be labelled, otherwise `#IMPLIED`,
6. a closing tag >

XML/Internal linking/References

For the references, use

1. an opening tag `<!ATTLIST`,
2. the name of the referring element,
3. the names of the reference attributes separated by spaces,
4. the keyword `IDREF`,
5. the keyword `#REQUIRED` if the element must always carry a reference, otherwise `#IMPLIED`,
6. a closing tag >

XML/Internal linking (cont)

For example,

```
<?xml version='1.0'?>
<!DOCTYPE map [
  <!ATTLIST country code   ID    #REQUIRED>
  <!ATTLIST country name   CDATA #REQUIRED>
  <!ATTLIST country border IDREF #IMPLIED>
]>
<map>
  <country code="uk" name="United Kingdom" border="ie"/>
  <country code="ie" name="Ireland" border="uk"/>
</map>
```

XML/Comments

It is possible to include comments in an XML document. They are made of

1. an opening tag "<!--",
2. some text without the sequence "--",
3. a closing tag "-->".

For example

```
<p>Our store is located at</p>
<!- <address>Eunpyeong-gu, Seoul</address> ->
<address>Gangnam-gu, Seoul</address>
```

Contrary to programming languages, comments are **not** ignored by the parsers and are nodes of the XML tree.

XML/Namespaces

Each XML document defines its own element tags, we can call its *vocabulary*.

In case we use external entities which refer to other XML document using, by coincidence, the same tags, we end with an ambiguity in the master document.

A good way to avoid these name clashes it to use **namespaces**. A namespace is a user-defined annotation of each element tag names and attribute names.

Therefore, if two XML documents use two different namespaces, i.e., two different tag annotations, there is no way to mix their elements when importing one document into the other, because each element tag carries an extra special annotation which is different.

XML/Namespaces (cont)

The definition of a namespace can be done at the level of any element by using a special attribute with the following syntax:

```
xmlns:prefix = "URL"
```

where *prefix* is the space name and *URL* (*Universal Resource Location*) points to a web page describing in natural language (e.g., in English) the namespace.

```
<?xml version="1.0"?>
<syllabus:journal
 xmlns:syllabus="http://konkuk.ac.kr/~rinderkn/syllabus.html">
 <syllabus:date>26 August 2006</syllabus:date>
 <syllabus:subject syllabus:hard="no">
   XML and company</syllabus:subject>
 <syllabus:abstract>
   We will study XML, XPath and XSLT.</syllabus:abstract>
</syllabus:journal>
```

XML/Namespaces (cont)

The scope of a namespace, i.e., the part of the document where it is usable, applies to the subtree whose root is the element declaring the namespace.

By default, if the prefix is missing, the element and all its sub-elements without prefix belong to the namespace. So, the previous example could be simply rewritten

```
<?xml version="1.0"?>
<journal xmlns="http://konkuk.ac.kr/~rinderkn/syllabus.html">
 <date>26 August 2006</date>
 <subject hard="no">XML and company</subject>
 <abstract>We will study XML, XPath and XSLT.</abstract>
</journal>
```

Note that the colon is missing in the namespace attribute.

XML/Namespaces/Example

An example of avoided clash name. File `fruits.xml` contains HTML code:

```
<table>
  <tr>
    <td>Bananas</td>
    <td>Oranges</td>
  </tr>
</table>
```

File `furniture.xml` contains a description of pieces of furniture:

```
<table>
  <name>Round table</name>
  <wood>Oak</wood>
</table>
```

XML/Namespaces/Example (cont)

The master document `main.xml` includes both files:

```
<?xml version="1.0"?>
<!DOCTYPE eclectic [
  <!ENTITY part1 SYSTEM "fruits.xml">
  <!ENTITY part2 SYSTEM "furniture.xml">
]>
<eclectic>
  &part1;
  &part2;
</eclectic>
```

The problem is that `table` has a different meaning in the two included files, so they should not be confused: this is a clash name.

XML/Namespaces/Example (cont)

The solution consists in using two different namespaces. First

```
<html:table xmlns:html="http://www.w3.org/TR/html4/">
  <html:tr>
    <html:td>Bananas</html:td>
    <html:td>Oranges</html:td>
  </html:tr>
</html:table>
```

Second

```
<f:table xmlns:f="http://www.e-shop.com/furnitures/">
  <f:name>Round table</f:name>
  <f:wood>Oak</f:wood>
</f:table>
```

XML/Namespaces/Example (cont)

But this is a heavy solution... Fortunately, namespaces can be defaulted:

```
<table xmlns="http://www.w3.org/TR/html4/">
  <tr>
    <td>Bananas</td>
    <td>Oranges</td>
  <tr>
</table>
```

Second

```
<table xmlns="http://www.e-shop.com/furnitures/">
  <name>Round table</name>
  <wood>Oak</wood>
</table>
```

XML/Namespaces/Example (cont)

The two kinds of tables can be mixed. For example

```
<mix xmlns:html="http://www.w3.org/TR/html4/"
     xmlns:f="http://www.e-shop.com/furnitures/">
<html:table>
...
  <f:table>
...
  </f:table>
...
<html:table>
</mix>
```

Note that element `mix` has no namespace associated (it is neither `html` nor `f`).

XML/Namespaces/Unbinding and rebinding

It is possible to unbind or rebind a prefix namespace:

```
<?xml version="1.1"?>
<x xmlns:n1="http://www.w3.org">
  <n1:a/> <!-- valid; the prefix n1 is bound to
               http://www.w3.org -->
    <x xmlns:n1="">
      <n1:a/> <!-- invalid; the prefix n1 is not bound here -->
      <x xmlns:n1="http://www.w3.org">
        <n1:a/> <!-- valid; the prefix n1 is bound again -->
      </x>
   </x>
</x>
```

XML/Namespaces/Unbinding the default namespace

```
<?xml version='1.0'?>
<Beers>
  <table xmlns='http://www.w3.org/1999/xhtml'>
    <!-- default namespace is now XHTML -->
    <th><td>Name</td><td>Origin</td><td>Description</td></th>
    <tr>
      <!-- Unbinding XHTML namespace inside table cells -->
      <td><brandName xmlns="">Huntsman</brandName></td>
      <td><origin xmlns="">Bath, UK</origin></td>
      <td><details xmlns="">
            <class>Bitter</class>
            <hop>Fuggles</hop>
            <pro>Wonderful hop, good summer beer</pro>
            <con>Fragile; excessive variance pub to pub</con>
          </details></td>
    </tr>
  </table>
</Beers>
```

XML/Namespaces/More name scoping

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- initially, the default namespace is "books" -->
<book xmlns='http://loc.gov/books'
    xmlns:isbn='http://isbn.org/0-395-36341-6'
    xml:lang="en" lang="en">
  <title>Cheaper by the Dozen</title>
  <isbn:number>1568491379</isbn:number>
  <notes>
    <!-- make HTML the default namespace
         for a hypertext commentary -->
    <p xmlns='http://www.w3.org/1999/xhtml'>
        This is also available
        <a href="http://www.w3.org/">online</a>.
    </p>
  </notes>
</book>
```

XML/Namespaces/Attributes

For example, each of the bad empty-element tags is invalid in the following:

```
<!-- http://www.w3.org is bound to n1 and n2 -->
<x xmlns:n1="http://www.w3.org"
   xmlns:n2="http://www.w3.org" >
  <bad a="1"    a="2"/>     <!-- invalid -->
  <bad n1:a="1" n2:a="2"/>  <!-- invalid -->
</x>
```

XML/Namespaces/Attributes (cont)

However, each of the following is valid, the second because **the default namespace does not apply to attribute names**:

```
<!-- http://www.w3.org is bound to n1 and is the default -->
<x xmlns:n1="http://www.w3.org"
   xmlns="http://www.w3.org" >
  <good a="1" b="2"/>    <!-- valid -->
  <good a="1" n1:a="2"/> <!-- valid -->
</x>
```

XML/Namespaces (cont)

Namespaces will be very important when learning XSLT.

Although namespaces are declared as attributes, they are present in the XML tree corresponding to the document as a special node, different from the attribute nodes.

XML/Processing instructions

In some exceptional cases, it may be useful to include in an XML document some data that is targeted to a specific XML processor.

This data is then embedded in a special element, and the data itself is called a **processing instruction** because it tells to a specific processor, e.g., Saxon, what to do at this point.

The syntax is

`<?target data?>`

The *target* is a string supposed to be recognised by a specific XML processor and the *data* is then used by this processor. Note that the data may be absent and that it contains attributes. For example:

```
<?xml version="1.0"?>
```

XML/Checking the well-formedness

All XML processors must check whether the input document satisfy the *syntactical* requirements of a well-formed XML document.

In particular,

- element tags must be closed, except for empty elements (this has to be contrasted with HTML),
- the predefined entities must be really predefined (unicodes are checked),
- internal entities must be declared in the prolog, etc.

Validating processors also check that the external entities are found (their well-formedness is checked after they have been inserted in the master document).

XML/Checking the well-formedness (cont)

There are several XML parsers available for free on the internet, implemented in several languages.

Most of them are actually libraries (API) for the programmer of an XML-handling application would need to link with.

The textbook provides a very basic standalone parser, `dbstat.pl`, written in Perl, which provides also some statistics about the document (like the number of elements of different kinds).

There is another, more complete, parser called `xmllint`.