XML Introduction

XML, Extensible Markup Language is a framework for defining markup languages. There is no fixed collection of markup tags in XML. Each XML language is targeted at a particular application domain, but the languages will share many features:

framework

→ they all use the same basic markup syntax

Syntax

→ they all benefit from a common set of generic tools for processing documents.

Generic tools

XML is not a single markup language that can be extended for other users, but rather it is a common notation that markup languages can build upon. XML is not an extension of HTML, nor is it a replacement for HTML, which ideally should be just another XML language. W3C has designed XHTML as an XML variant of HTML. One way to make use of such a XML document is to specify a stylesheet that defines a meaning of the markup tags in terms of instructions for rendering the document on the screen. XML is inherently internationalized and platform independent. All XML documents are written in the Unicode alphabet. XML is intended to be the future of all structured information. This includes information stored in relational databases, which has motivated the development of the powerful query language, XQuery

Variant of HTML

Stylesheet

Unicode alphabet

XQuery

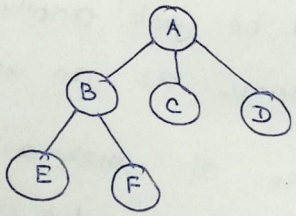
Design of an XML Document

The development of XML began in the mid-90s and in November 1996, the initial XML draft was produced as a pure subset of SGML. In February 1998, XML 1.0 became a W3C recommenda The main new features of XML 1.1 are support for recent and future changes in the Unicode standa and a notion of normalization of character encodings

SGML → XML

XML Trees

An XML document is a hierarchical structur called an XML tree, which consists of nodes of various kinds arranged as a tree.



A tree

Nodes are drawn as circles.

→ The topmost node is called a root.

→ The edges show the parent-child relationship between the nodes.

Eg) B is a child of A and A is the parent of B.

→ The content of a node is the sequence of its child nodes.

→ For the node A, the content is the sequence (B, C, D).

→ The nodes in a tree have different numbers of children

→ Nodes with no children are called leaves.

→ E, F, C and D are leaves.

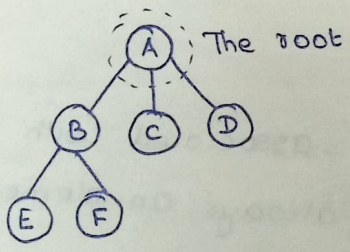
An XML tree is ordered meaning that the ordering of children of a node generally is significant.

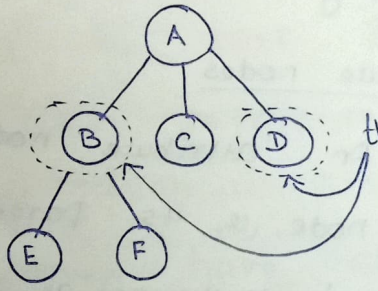
→ The siblings of a node are the other children of the parent of the node.

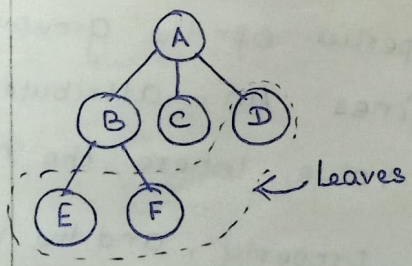
→ The ancestors of a node consist of its parent, the parent of the parent and so on.

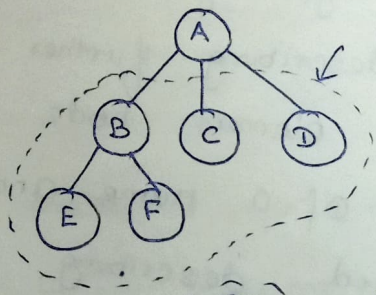
→ The descendants of a node is the set consisting of its children, the children of the children and so on.

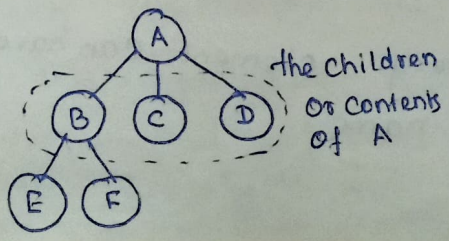
Eg) The descendants of the root node is the set of all nodes in the tree, except the root itself.

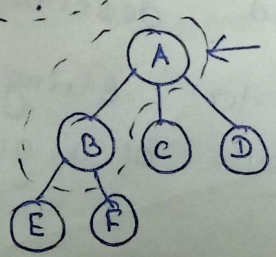












In the XPath data model, an XML tree is a special kind of ordered tree whose nodes can be of the following kinds:

text nodes:

A text node corresponds to a fragment of the actual information being represented by the XML document. Every text node is labeled with a nonempty text string containing this information.

Text nodes have no children. They are leaves in the tree.

Two text nodes cannot occur as siblings of each other, except if another kind of node appears in between.

Element nodes

An element node defines a logical grouping of the information represented by its descendants. Every element node has a name, a word that describes the grouping.

Attribute nodes

An attribute node is associated with an element node, ie, its parent is always an element. Attributes typically act as refinements of the element's name describing further properties of the grouping that the element node defines. An attribute is a pair of a name and a value, where the name is a word describing the property, and the value is some text string. Every element can have at most one attribute of a given name.

Comment nodes:

A Comment node is a special leaf node labeled with a text string. One should think of Comment nodes as Comments in programming languages: they contain informal meta-information that most tools simply ignore.

Processing Instruction nodes

Target and value.

A processing instruction node has a target and a value, and can be used to convey specialized meta-information to various XML processing tools. The target is a word that specifies the kind of processing tool that the processing instruction is directed toward; all other tools can ignore it. The value is a text string containing the relevant meta-information to the tool.

Eg) The target could be xml-stylesheet, which is recognized by XSLT processors and the value a URI reference to an XSLT stylesheet used by such a processor. Processing instructions never have child nodes.

root nodes

Root Nodes

Every XML tree starts with a single root node, which represents the entire document. The children of the root node consists of any number of Comment and processing Instruction nodes together with exactly one element node, which is called the root element.

The Parent-Child relationship in XPath data model is not symmetric.

The children of an element is an ordered sequence containing text, elements, comments and processing instruction. In addition, the element is associated with a set of attribute nodes.

Well Formed XML Document

XML syntax requires a single root element, a start tag and end tag for each element and properly nested tags.

XML is case sensitive, so the proper capitalization must be used in elements. A document that conforms to this syntax is a well-formed XML document and is syntactically correct.

If an XML parser can process an XML document successfully, that XML document is well formed. Parsers can provide access to XML-encoded data in well-formed documents only.

Valid XML Documents

An XML document can reference a Document Type Definition (DTD) or a schema that defines the proper structure of the XML document. When an XML document references a DTD or a schema, some parsers (called validating parsers)

Can read the DTD/schema and check that the XML document follows the structure defined by the DTD/schema. If the XML document conforms to the DTD/schema (ie, the document has the appropriate structures), the XML document is valid.

Note: By definition, a valid XML document is well formed.

DTD/schema.
↳. Document gets processed.

DTD - Document Type Definition

XML has the first working draft contained a built-in Schema language: Document Type Definition (DTD). DTD is a reasonably simple schema language with a rather restricted expressive power. It has provided the starting point for the development of newer and more expressive schema languages.

i) Document Type Declarations

An XML document may contain a document type declaration, which is essentially a reference to a DTD schema. By inserting such a declaration, the author states that the XML document is intended to be valid relative to that schema. In this way, documents become self describing, which makes it easy for tools to determine what kind of input they receive.

A document type declaration typically has the form

<!DOCTYPE root SYSTEM "URI">

where

root is an element name

and URI, called the system identifier is a URI of the DTD schema.

The document type declaration appears between the XML declaration (<?xml ...?>), if present, and the root element is the instance document. The instance document is valid if the name of its root element is root and the document satisfies all constraints specified in the DTD schema as described below.

Assume that we have written a DTD Schema for RecipeML and made it available at the URL http://www.brics.dk/ixwt/recipes.dtd. Our recipe collection could then look as follows (where the content of the root element is shown as '...')

```
<?xml version="1.1"?>
<!DOCTYPE Collection
    SYSTEM "http://www.brics.dk/ixwt/recipes.dtd">
<Collection>
  ...
</Collection>
```

As a supplement to the system identifier, a document type declaration may contain a public identifier, which is an alternative way of specifying the DTD schema.

Eg)

XHTML documents often contain the following declaration:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Here,

the string following PUBLIC is the public identifier and the string on the next line is the system identifier.

This specific public identifier is technically an unregistered identifier, it is owned by the W3C, it refers to a DTD for the 'XHTML 1.0 Transitional' language, and it is written in English.

The notion of public identifiers is a relic from SGML; Usually, public identifiers are simply ignored. However, they do come in handy in situations where the DTD schema may exist at many different locations.

Eg) W3C's HTML/XHTML validator reads the public identifier to determine the version of HTML or XHTML being used; it cannot determine the version from the system identifier since it is not required that the schema resides at a fixed location.

The URN mechanism provides a more general solution to the issue of naming resources without specific locations, and some systems allow URNs as system identifiers.

Consider the document type declarations of the form

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

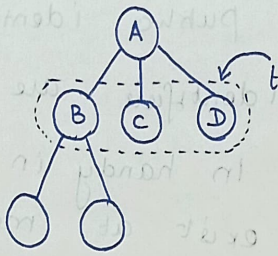
where

'...' consists of declarations of elements, attributes and so on.

Such internal declarations that appear within the instance document have exactly the same meaning as if they were moved to a separate file being referred to with a system identifier. The document type declaration may even contain a mix of internal and external declarations. It is a much better idea to keep the DTD schema separately from the instance documents, such that many instance documents can share the same schema.

DTD Schemas may contain comments using the same notation as in XML:

<!-- this is a comment -->



Comments have no formal meaning, but they are often used in DTD schemas to explain extra restrictions that DTD is unable to express formally.

Eg)

In the DTD schema for XHTML, one may find a comment like this

<!-- anchors shouldn't be nested -->

DTD Schemas contain processing instructions.

Element Declarations

A DTD schema consists of declarations of elements, attributes and various other constructs.

An element declaration looks as follows:

<!ELEMENT element-name content-model>

where

element-name is an element name, such as table or ingredient and content-model is a description of a content model, which defines the validity requirements of the contents of all elements of the given name.

Every element name that occurs in the instance document must correspond to one element declaration in order for the document to be valid. Moreover, the contents of the element must match the associated content model as defined below.

Content models come in four different flavours:

empty: If the content model of an element is EMPTY, then the element must be empty. Being empty means that it has no contents, but this says nothing about attributes.

any: The content model ANY means that the contents of the element can consist of any sequence of character data and elements.

Each of these elements must be declared by corresponding element declarations. The ANY content model is mostly used during development of DTD schemas for elements ~~that have not yet been~~ deser

Mixed Content: A content model of the form

(#PCDATA|e1|e2|...|en)*

where each ei is an element name means that the contents may contain arbitrary character data, interspersed with any number of elements of the specified names.

element content: To specify constraints on the order and number of occurrences of child elements, a content model can be written using the following variation of the regular expression notation.

→ the alphabet consists of all element names

→ Concatenation is written with comma (,) instead of using juxtaposition of the operands.

→ Only a restricted form of regular expressions called deterministic regular expressions is permitted, which makes it easier to check whether or not a sequence of element names matches an expression.

For the contents of an element with such a content model to be valid, it must match the regular expression.

The various constructs are summarized below:

Construct	Meaning
EMPTY	empty Contents
ANY	any Contents
#PCDATA	Character data
element name	an element
,	Concatenation
|	Union
?	Optional
*	Zero or more repetitions
+	One or more repetitions

Comments and Processing Instructions occur anywhere in the instance document;

→ their use cannot be constrained except with EMPTY since 'empty contents' means 'no elements, Character data, Comments or processing instructions whatsoever!

→ Considering the design of the DTD language

i) Either arbitrary character data is permitted in the Contents or no character data is permitted.

(ii) If character data is to be permitted in the contents then we have no choice but using the mixed content model, which cannot constrain the order and number of occurrences of the child elements.

SGML and XML languages can be classified as either document oriented or data oriented depending on whether they use the mixed content model or not.

(iii) Attribute-List Declarations

An attribute-list declaration has the following form:

<!ATTLIST element-name attribute-definitions>

where element-name is an element name and attribute-definitions is a list of attribute definitions (name, type, declaration), each having three constituents

i) attribute-name
ii) attribute-type
iii) default-declaration

The most important categories of attribute types are the following:

String type: The attribute type CDATA (for character data) means that the attribute can have any value.

Enumeration: An attribute type of the form

$(s_1 | s_2 | \ldots | s_n)$

where each s_i is some string, means that the value of the attribute must be among the strings $s_1, s_2, \ldots, s_n$

Name tokens: The attribute type NMTOKEN means that the attribute value must be a name token.

In XML 1.1, almost all characters are permitted in name tokens except those which are delimiters. However hyphens, underscores, colons and periods are explicitly permitted

The variant NMTOKENS denotes a whitespace separated nonempty list of name tokens.

identity/reference type: The attribute type ID means that the value of the attribute uniquely identifies the element containing the attribute (ie, No two attributes of type ID can have the same value in an XML document. Only one attribute of type ID is permitted per element.

ID IDREF

The attribute type IDREF is used for references to elements with an ID attribute. The value of an attribute of type IDREF must match the value of some attribute of type ID in the same document.

The attribute type IDREFS is like IDREF but permits multiple references as attribute values.

For all attribute types other than CDATA, the attribute value is

→ before the actual validation takes place

→ Normalized by discarding any leading and trailing whitespace and replacing sequences of whitespaces by a single space character.

Discarding leading, Trailing, Replacing sequence of whitespace, Single whitespace

The CDATA type is used for both the maxlength and the tabindex attribute in input elements in XHTML.

```
<!ATTLIST input maxlength CDATA #IMPLIED
             tabindex  CDATA #IMPLIED>
```

These attributes are used to specify the maximal number of characters in a text input field and the field position in tabbing order, respectively.

The authors of the DTD Schema for XHTML have inserted the informal Comment.

<!-- One or more digits - ->

at that place in the Schema as an attempt to make it clear to the human reader that not all values make sense.

The NMTOKEN type is used for certain name attributes in XHTML:

<! ATTLIST form name NMTOKEN #IMPLIED ...>

This declaration shows that name = "my.form", name = "87" and name = " 87 " are all valid in form elements, whereas name = "myform" and name = " " are invalid.

The ID and IDREF(S) types can be used as a intra-document reference mechanism, for denoting keys when XML documents are used as databases, and for specifying anchors for easy addressing into documents.

Eg) RecipeML

the id attribute of recipe element has type ID, and the ref attribute of related elements has type IDREF.

<! ATTLIST recipe id ID #IMPLIED>
<! ATTLIST related ref IDREF #REQUIRED>

The following RecipeML document is then valid

<collection>
<description> My valid Recipe Collection </description>
<recipe id = "r101"
Recipe 1
<title> Beef Parmesan with Garlic Angel Hair Pasta </title>

...

<related ref = "r103">

this goes well with Linguine Pescadaro

</related>

</recipe>

<recipe id = "r102">

<title> Ricotta pie </title>

...

</recipe>

<recipe id = "r103">

<title> Linguine Pescadoro </ title>

...

</recipe>

</Collection

Whereas the following is invalid because it contains two ID attributes with the value r101 and also an IDREF attribute whose value r12345 does not match that of an ID attribute.

<Collection>

<description> My Invalid Recipe Collection </ description>

<recipe id = "r101">

<title> Beef Parmesan with Garlic Angel Hair Pasta Recipe </title>

...

<related ref = "r12345"> Spiced Beef Stew is also great </related>

</recipe>

```
<recipe id = "r101">
    <title> Ricotta Pie </title>
    ...
</recipe>
<recipe id = "r113">
    <title> Linguine Pescadoro </title>
    ...
</recipe>
</Collection>
```

→ An IDREF attribute may be thought of as a special pointer from its containing element to the element with the corresponding ID attribute.

→ In addition to the main attribute types, the DTD language contains three rather obscure types:

1) ENTITY
2) ENTITIES
&
3) NOTATION

→ The third constituent of an attribute declaration - the default declaration - specifies whether the attribute is required or optional and potentially also a default value.

→ The following kinds are possible:

Required: #REQUIRED means that the attribute must be present.

Optional: #IMPLIED means that the attribute is optional. No default is provided if the attribute is absent.

Optional, but default provided: "value", where value is a legal attribute value, means that the attribute is optional, but if it is absent, this value is used as a default.

fixed: # FIXED "value" means that the attribute is optional.

→ If it is absent, then this value is used as a default
→ If it is present, then it must have this specific value

The following declaration from the DTD schema for XHTML illustrates the first three variants:

```
<!ATTLIST form
    action      CDATA   #REQUIRED
    onsubmit    CDATA   #IMPLIED
    method      (get|post)  "get"
    enctype     CDATA   "application/x-www-form-urlencoded">
```

This declaration shows that the action attribute in form elements is required, and the onsubmit attribute is optional

The method attribute is also optional, but if it is omitted, the default value get is inserted by the DTD processor, and similarly for the enctype attribute.

A DTD processor will then validate the following part of an instance document.

```
<form action = "http://www.brics.dk/ixwt/examples/hello.jsp">
...
</form>
```

and normalize it as follows:

```
<form action = "http://www.brics.dk/ixwt/examples/
                                        hello.jsp"
      method = "get"
      enctype = "application/x-www-form-urlencoded">
...
</form>
```

When more than one attribute-list declaration is provided for a given element name, the declarations are merged.

If an attribute of a given name is specified more than once for an element, then the first one takes effect, but friendly XML parsers issue a warning if this situation occurs

A typical use of #FIXED is to fake namespace declarations, as in the DTD schema for XHTML:

```
<!ATTLIST html xmlns CDATA #FIXED "http://www.w3.org/
                                        1999/xhtml">
```

iv Conditional Sections, Entity and Notation Declarations

~~In existing~~

Internal entity declarations constitute a poor man's macro mechanism.

If our DTD schema contains

```
<!ENTITY Copyrightnotice "Copyright &#169; 2005 Widgets 'R' us">
```

then the DTD processor may convert the following fragment of an instance document

A gadget has a medium size head and a big gizmo Subwidget. & Copyright notice

into

A gadget has a medium size head and a big gizmo Subwidget © 2000 Widgets 'R' Us.

The predefined entities are implicitly defined using internal entity declarations. Another declaration can be added:

<!ENTITY Copy "©">

So that we can write

&Copy

which is easier to remember than © whenever we want a © symbol.

An internal parameter entity declaration is a macro definition that is only applicable within the DTD schema, and not in the instance document.

Eg) As in XHTML, it is defined as

<!ENTITY % Shape "(rect|circle|poly|default)">

Can and then references are used within the Scheme such as in

<!ATTLIST area shape %Shape; "rect">

which is then equivalent to

<!ATTLIST area shape (rect|circle|poly|default) "rect">

The % symbol in the declaration and the reference indicates a parameter entity, whereas & is used for normal entity references.

An external entity declaration is a reference to another resource, which consists of XML or non-XML data. A reference to another XML file may be declared by

<!ENTITY widgets SYSTEM "http://www.brics.dk/ixwt/widgets.xml">

Occurrences of the entity reference &widgets; will then result in the designated XML data being inserted in place of the entity reference, via an entity reference node.

A reference to a non-XML resource, called an unparsed entity, can be declared as in

<!ENTITY widget-image
SYSTEM "http://www.brics.dk/ixwt/widget.gif"
NDATA gif>

Here, the NDATA part refers to a notation, which describes the format of an unparsed entity, for example by the declaration,

<!NOTATION gif
SYSTEM "http://www.iana.org/assignments/media-types/image/gif">

which could be used as a description of GIF images. Notation declarations may also be used for description of processing instruction targets.

It does not make sense for an instance document to contain entity references to unparsed entities. These entities are used in conjunction with the special attribute types ENTITY and ENTITIES:

→ the value of an attribute of type ENTITY must match the name of an unparsed entity,

→ ENTITIES must match a WhiteSpace separated non empty list of such names.

When an XML parser that performs validation with DTD encounters such attributes in the instance document, it informs the application of the associated SYSTEM/PUBLIC identifiers and notation names.

Notations may also be used directly in attributes: the value of an attribute of type NOTATION must match the name of a notation declaration. This can be used to describe formats of non-XML data.

Finally, Conditional sections allow parts of the schema to be included or excluded (via) a switch.

The typical use of this mechanism is to let an external DTD schema contain a number of declarations and then select only some of them for use in the instance document. Conditional sections are typically combined with the parameter entity mechanism.

Example:

An external DTD schema may contain

```
<![%person.Simple;[
    <!ELEMENT person(firstname, lastname)>
]]>
<![%person.full;[
    <!ELEMENT person(firstname, lastname, email+, phone?)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT phone (#PCDATA)>
]]>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
```

In the internal DTD schema, we can define either

```
<!ENTITY % person.Simple "INCLUDE">
<!ENTITY % person.full "IGNORE">
```

to select the Simple version of the person element, or

```
<!ENTITY % person.Simple "IGNORE">
<!ENTITY % person.full "INCLUDE">
```

to get the full version.

The parameter entity references expand to the keywords IGNORE and INCLUDE, which result in respectively, disabling and enabling the contents of the blocks.

Checking Validity with DTD

A DTD processor works by first parsing the instance document, that is constructing its XML tree representation, and the DTD schema, including all external schema subsets. Parsing succeeds if the document is well-formed.

Note:

The DTD language does not use an XML notation, a different parsing technique is needed for parsing the DTD Schema.

The DTD processor checks that the name of the root element of the instance document is correct. In the next phase, most of the actual validation work is performed in a simple traversal of the XML document:

For each element node, the processor looks up the associated element declaration and attribute-list declarations, and

1) Checks that the content of the element node matches the content model of the element declaration.

2) normalizes the attributes according to the declarations, which consists of insertion of default attributes and pruning of whitespace where applicable.

3) Checks that all required attributes are present and

4) Checks that the values of the attributes match the associated attribute type.

Entity references are either expanded during the traversal or kept in the tree as entity reference nodes In either case, the validation treats them as if they were expanded. In the same traversal, all ID attributes are collected and checked for uniqueness.

After this phase, the only thing that remains is the check for IDREF and IDREFS attributes. For each of these, the processor checks that each reference corresponds to some ID

Finally, if no validation errors were detected, the processor outputs the normalized instance document, either in its textual form or in its tree representation to the application.

Naturally, a DTD processor is not required to follow this exact sequence of phases as long as it has the same resulting behavior.

Instance documents that do refer to external DTD declarations but do not rely on attribute defaults and entity declarations from these external parts may contain a standalone declaration in the XML declaration:

<?xml version = "1.1" standalone = "yes" ?>

This declaration tells the XML processors that if they are only interested in parsing the document but not in checking validity, then the external DTD declarations can safely be ignored.

Checking validity with DTD: A DTD processor works by first parsing the instance document, ie, constructing its XML tree representation and the DTD schema, including all external schema subsets. Parsing succeeds if the document is well formed.

Limitations of DTD

1. DTD cannot constrain character data. What is needed is a more powerful datatype mechanism for describing character data.

Datatype

2. DTD cannot specify that the value of a particular attribute must be, for example,

Value

- → an integer
- → a URI
- → a date
- → or whatever data types we might use.

3. Element and attribute declarations are entirely context insensitive.

4. Character data cannot be combined with the regular expression content model. This means that if we need to permit character data in the contents, then we cannot control the order of elements or their number of occurrences.

5. The content models lack an 'interleaving operator.

6. DTD provides very limited support for

→ modularity

→ reuse

and

→ evolution of schemas

7. The normalization features in DTD are limited. Although there is a default mechanism for attributes, there is none for element contents. Also, using the special xml:space attribute does not cause any removal of insignificant whitespace in character data by the DTD processor – it only indicates the intended meaning to the application.

8. DTD does not permit embedded structured self-documentation. Comments are allowed, but they cannot contain markup.

9. The ID/IDREF mechanism is too simple. It is desirable to be able to specify a more restricted scope of uniqueness for ID attributes than the entire instance document. It is also inconvenient that only individual attribute values can be used as keys. It would be more useful if the key could consist of multiple attribute values or even character data.