Topic II is about markup languages. Markup languages are “one level above” simple encodings, since they allow extra-textual elements to be included in the encoding. A simple example is the use of an emoticon in an e-mail. When you interpret a “:-)” as an emoticon, you are essentially recognizing some form of extra-textual markup. In this case, it is special combinations of ASCII characters that have pragmatic, conversational meaning. You may not have thought of emoticons as text markup, but this is essentially what is going on.

Two questions to keep in mind as you work through this topic are as follows:

1. Where does encoding end and markup begin?
   - Encoding takes place with respect to the alphabet symbols of the input language. Markup, on the other hand, uses these same symbols (or combinations of them) not as symbols of the input language, but as special codes, which may mean anything from how the data is to be displayed to what the data means within the larger text itself.
   - The origin of the World Wide Web began with the development of a text markup language, HTML. HTML uses the simple ASCII (or UTF8) encoding. Yet, using special sequences of the usual English characters, we can specify both logical and visual properties of the data.
   - A key point to think about concerning HTML is what happens when the special sequences themselves need to be represented. For example, the sequence of characters “<html>” has special meaning in HTML. But what if we want to write that particular sequence as an English “word”? (For example in a text about HTML.) We need special “escape sequences” which tell the interpreter (the web browser) that we are speaking literally now – so we write “&lt; html &gt;” to get “<html>” to display on the screen. This is essentially a phenomenon that occurs when encoding and markup meet.

2. Where does markup end and processing begin?
   - Data markup occurs because we include special codes within the representation of the data itself. In contrast, processing occurs when the data is changed, manipulated, or displayed and this involves patterns not in the original data at the representation level. One common example is syntax highlighting. This is a
feature of many programming environments, and involves displaying keywords in a special color (for example). The program itself remains plain text, however. There is no special markup added to the program which distinguishes keywords. Rather, a parser finds these keywords and displays them in color according to the rules of the programming language.

- One fly in the ointment with the seemingly clean distinction made above, however, occurs with binary executable files. As we will see later in the course, computers “run” sequences of bits. A binary file executes directly, and the bits within the file determine the processing. So where is the distinction between encoding, markup, and processing with a binary executable?

Often, the term *markup language* has a specific meaning, restricted to such text markup languages as XML and [X]HTML (the SGML descendents). We will look at these widely used markup languages in more detail below (§2 and §3). However, in this course, we expand the idea of a markup language and consider *data markup* in a more general sense, as self-referential information contained in a data file beyond the pure data itself. Once we have this general outlook, we can classify markup into several meaningful categories (§1).

## 1 Classification of Data Markup

We can classify various data markup schemes by their intent. In this case, data markup falls into four broad classes: *pragmatic*, *logical*, *visual*, and *semantic* markup.

All of these categories apply almost solely to text documents. Markup is usually not used with numbers, binary executables, or (more generally) data that requires specific, known processing techniques. In contrast, markup is an attempt to incorporate some type of intelligence into the vast amounts of human-readable text available in digital form. The computer cannot “read” this text like you or I; rather, every bit of processing it performs on the text must be spelled out in great detail. Markup languages allow us to explicitly tag the text with some of the more “obvious” functionality.

The four types of markup are discussed in the following sections.

### 1.1 Pragmatic Markup

*Pragmatic Markup* is used in discussion forums, newsgroups, chat rooms, e-mail and other forms of electronic communication. Emoticons fall into this category. Pragmatic markup is essentially agreed-on extensions to natural language that describe facets of communication not contained in the text itself. Some common acronyms or punctuation uses also have this purpose (such as ROTFL and >grin<). Pragmatic markup uses plain text to annotate linguistic communication with the *pragmatic* features of everyday communication, such as shrugging one’s shoulders, or grinning. These are non-linguistic actions that are incorporated into the text.

Some applications recognize pragmatic markup and change the text display accordingly. For example, the mail editor used with Oncourse recognizes common emoticons and changes them into picture icons. This seems like a nice feature, until you try to send
programs via the Oncourse mail – the string ;) (which is common in languages with a Java-like syntax) changes to a winking happy face, obscuring the original code.

1.2 Logical Markup

Logical markup is often thought of as “pure” markup, and was what HTML was originally intended to do. The idea here is that rather than focus on visual features of a document’s contents (for example, whether a piece of text is bold or not), the markup contains directives governing the logical structure of the text as a whole.

A written document very often has some or all of the following features:

- A title,
- an outline structure (including several levels of nested headings),
- lists, itemized or enumerated,
- tables,
- figures and/or pictures,
- etc.

Probably you recognize that these document elements map directly to HTML tags.

A reasonably clear case of logical markup versus visual markup (see next section) in HTML occurs is when inline text is emphasized. You are using logical markup if you write "<em>...</em>" in HTML, but you are using visual markup if you write "<i>...</i>". In the former case, you are tagging a piece of text as having some kind of emphasis compared to the surrounding text. It is up to the computer application how this actually displays – bold, italics, surrounded with underscores, etc. In the latter case, you are tagging a piece of text so it must display in italic font. Even today, some browsers use plain text and have no italic font. In the case of logical markup, you’re giving these browsers a choice, as long as they convey the logical structure of your text; in the case of visual markup you’re specifying exactly how you want the text to display, without committing yourself to a particular logical function.

1.3 Visual Markup

One of the most widespread uses of data markup is visual markup. Visual markup is useful for expressing how the data should display in any application associated with that data. Visual markup (unlike, for example, syntax highlighting) requires minimal parsing or processing of the data; rather the data itself specifies how it is to be displayed.

HTML is most commonly used as a visual markup tool, even though the language was developed with logical markup in mind. In particular, tables are often used as devices to attain a particular visual layout of the web page, which is in opposition to their intended use for structured data comparison. This use of HTML (and tables in particular) is sometimes disparaged, however it may seem natural to you (and most web designers use HTML for page layout).
Essentially, visual markup involves tagging specific parts of the data with display instructions. An example is “<i>googlebot</i>” in HTML – this is a directive to display the word “googlebot” in italic font. Visual markup can become quite complicated, for example using HTML tables to display site tables of contents and advertisements.

Beyond HTML, most productivity applications involve some form of visual markup. This is reflected in the proprietary markup languages “.doc” (MS Word), “.xls” (MS Excel), and “.ppt” (MS PowerPoint), as examples. If you think about it, something in the actual Word document needs to say “this text is bold”, “this text is indented half an inch”, etc. These are visual formatting commands, and are part of the MS Word filetype. Of course, there are also logical markup directives in a “.doc” file, such as specifying that a particular piece of the document is from a previous revision (so is stored on disk, but not displayed in the application window).

In general, visual markup occurs when specific directives are inserted into the data which determine how a specific application is to display the data. No commitment is made about the logical or pragmatic (or semantic) implications of the data, rather these are just “pretty printing” directives.

Question 1.1
Research the PostScript markup language on the web. Is PostScript a visual or logical markup language (or both)? Explain why.

1.4 Semantic Markup

An exciting new application of markup languages has to do with the semantic web. The semantic web involves three categories of markup languages designed to incorporate true semantics into the data comprising the vast Internet. These languages are responsible for (1) annotating data with semantic terms (XML), (2) expressing relationships among data using ternary relations (RDF), and (3) encoding the basic assumptions necessary for correct inference (Ontologies).

The real impact of the latter two technologies is in how they are used. Thus, relationships and inference are an intrinsic part of the semantic web, but arise mostly out of the processing that is done in accordance with the stored rules. Pure semantic markup is in the XML that underlies the semantic web: this is an attempt to couple the atomic data with useful meaning. Read more about the semantic web in [2]. We will learn more about XML below (§3).

Question 1.2
The ideal of computers reading and understanding textual information is not new. Why might the semantic web succeed with this ideal where other techniques have (so far) failed? Discuss your answer.

2 XHTML

Once the Internet began to gain ground (and expand exponentially), it was quickly realized that simply having vast hoards of human-readable documents was not particularly useful for automatically finding nuggets of information. In the mid 80s, the markup language SGML was standardized to meet the need for tagging pure data with logical
meaning. This ideal is highly complex and multi-faceted, and in response SGML is highly
complex and onerous to learn. Because of this, it was simply not practical for the majority
of Internet users to use SGML.

In response, Tim Berners-Lee designed a small, easily learnable language based on
SGML he called HTML. You might agree that HTML is easy to learn and quite intuitive.
However, the ideal of logical (and semantic) markup was left behind with HTML. Most of
the text in a document remains unmarked, and people had widely varying ways of using
HTML to mark up their web pages. The demand for more complex visual formatting (as
the multimedia web came of age) required many extensions to HTML, which unfortu-
nately were not standardized. Web browsers became large and complicated applications
which defined their own brand of HTML. So the problem of finding nuggets of information
within the mass of data was further complicated by irrelevant visual layout markup.

Recently (late 1990s), a new descendant of SGML has emerged: XML. XML remains
true to the logical and semantic ideals of data markup. Yet, XML allows each corporation,
organization, or even individual user to make their own semantic and logical markup.
Amazingly, XML does this but still is standardized enough and simple enough that (given
a document model), useful querying and restructuring of the data can be performed.

But the Web is HTML. Thus, a hybrid language between XML and HTML emerged
in the past couple years, known as XHTML. XHTML conforms to the stricter XML
syntax, yet enables people to keep most of their existing HTML formatting. It is a
good idea to keep pace with the latest version of XHTML. In particular, you can go
to http://validator.w3.org and find out if your web page conforms to the XHTML
standard.

We will now look at the specifics of XHTML. It is assumed you already have experience
with HTML, so the focus is on how XHTML differs from HTML (§2.1) and a quick
review of the HTML tags you should know about (§2.2). Additionally, we will look at
the language CSS which associates visual markup with XHTML (§2.3).

2.1 HTML vs. XHTML

Here are the differences between HTML and XHTML you should know about:

1. Elements and attributes in XHTML must be in lower-case.

2. Every opening tag must have an end tag in XHTML, even empty tags. Furthermore,
   the opening and closing tags must be properly nested.

3. Attribute values in XHTML must always be single or double quoted.

4. Attribute minimization is not allowed in XHTML.

5. The HTML name attribute is deprecated in favor of the XHTML id attribute.

6. XHTML documents come associated with DTDs (Document Type Definitions)
   defining their structure.

7. XHTML documents use XML namespaces.

In general, the most striking difference between XHTML and HTML is that an XHTML
document must be well-formed and valid.
Definition 2.1
1. An XML document is well-formed if it conforms to the syntactic rules for XML documents (such as proper nesting and allowed tag identifiers). Well-formed documents do not have any inherent meaning or use associated with the elements within.

2. An XML document is valid if it validates against a particular document model. This document model describes the logical structures that are valid within the model (for example, a “subtitle” element must follow a “title” element).

In HTML, if a document had errors, it was just an HTML document with errors (which were usually displayed in the browser as plain markup text). In XHTML, this relaxation is not available. Either a document is XHTML, or it has syntax errors and it will not load. Perfect syntax is important in XHTML, as in programming languages. Furthermore, every XHTML document must validate against one of three published document models. We will now look at what this means in practice.

The Lower Case Rule All element and attribute names in XHTML must be lower case. In HTML, either upper or lower case can be used. XML, on the other hand, is case sensitive. Thus, it was decided to use all lower case for the XHTML keywords.

Tag Closing and Nesting In XHTML, every opening tag must have a corresponding closing tag. This extends even to tags which were “stand alone” tags in ordinary HTML. As examples:

- In HTML, you could omit the closing tags in some cases; for example <p> would start a new paragraph whether or not the previous paragraph ended with a </p> tag. In XHTML, every opening tag must have a corresponding end tag.

- In HTML, some tags do not have a natural corresponding end tag, for example <hr> (hard rule) and <br> (line break). In this case, we close the tag by including a />, as in: <hr /> or <br />.

Further to this, opening and closing tags must be properly nested. Thus, each closing tag must close the nearest opening tag. For example:

```
<!-- This is WRONG -->
<p>The quick brown <em>fox jumped over the lazy dog.</em></p>
```

```
<!-- This is RIGHT -->
<p>The quick brown <em>fox jumped over the lazy dog.</em></p>
```

Quoted Attributes In XHTML, all attribute values must have single or double quotes, including numeric values. For example (from [3]):

```
<!-- This is WRONG -->
<table align=center width=85% border=2 height=200>
```

```
<!-- This is RIGHT -->
<table align="center" width="85%" border="2" height="200">
```
No Attribute Minimization  Stand alone attribute values are not allowed in XHTML. Instead, the value must be repeated both as the attribute name and the attribute value, as in the following example:

```html
<!-- This is WRONG -->
<hr color="#900015" width="2" noshade />

<!-- This is RIGHT -->
<hr color="#900015" width="2" noshade="noshade" />
```

Use the id Attribute for Naming Elements  In HTML, certain elements use the name attribute to assign a name which can be used for later references to that element. You have probably seen this with the `<img>` and `<a>` elements. In XHTML, the name attribute is deprecated; rather the id attribute is preferred. For backward compatibility, it is usually a good idea to use both attributes, as in the following example.

```html
<img src="my_pic.jpg" id="Me" name="Me" alt="My picture" />
```

Structural Specification and Namespaces  Every XHTML document must validate against one of three available document models. We will see document models below (§3.3). For now, you can think of them as specifications as to what tags and attributes your XHTML document can use (and in what order). For example, all three XHTML models specify that the root element of your document must be the `<html>` element.

The three XHTML models provided by the W3C are:

1. **Strict**: This model defines strict XHTML 1.0. Any document that conforms to this model must not use any deprecated or frame elements. Furthermore, the markup must be purely logical (i.e. visual markup must be separated from the document structure). This is the version of XHTML you should strive to conform to for this class.

2. **Transitional**: If you document needs frames, you should use the transitional model. This model allows elements which are deprecated (and will disappear in future versions of XHTML).

3. **Frameset**: This model is even further removed from the goals of XML and XHTML and allows significant visual markup (that may bear no relation to the structure of the document). It is not recommended you use this model.

In addition to one of the 3 models provided by the W3C, your XHTML document must use an appropriate XML namespace. The namespace gives the terms and names which will be used both in the document model and in the XHTML document itself.

2.1.1 XHTML Skeleton

Putting the last few paragraphs together, here is an XHTML skeleton that utilizes the strict DTD and appropriate XML namespace, and includes the XHTML elements that will structure your XHTML document. It is a good idea to start with this skeleton whenever you create a new XHTML document.
2.2 Basic XHTML

In this section, we will quickly review the HTML tags that you should know about. Except where noted, these tags function identically in XHTML (you just have to be more careful about syntax errors). Keep in mind that you should use only the structural (logical) elements of HTML; for example, you should use `<em>...</em>` as opposed to `<i>...</i>`. The strict XHTML document model will enforce this, so you should check all your XHTML documents at [http://validator.w3.org](http://validator.w3.org).

Table 1 lists the basic elements and attributes you should be familiar with. If you are unfamiliar with any listed, make sure you look them up, for example on the web. This is not a comprehensive listing of the XHTML elements and attributes; rather it is a listing of the ones you should be familiar with (at a minimum) for this course. Not listed is the `<!-- ... -->` element, which defines a comment in XHTML (as in HTML).

Note that most attributes not listed are for visual formatting, and thus do not conform to the strict XHTML DTD. In general, the XHTML mantra states that XHTML markup should be purely logical (or semantic), and all visual formatting directives should be given by associated styles written in CSS (§2.3).

A Note on URIs The term URI stands for Uniform Resource Identifier. If you view every resource accessible via the entire World Wide Web as a point in a large space, the URIs are the coordinates of those points. You probably know about URLs (Uniform Resource Locators). URL is an informal term, and is now deprecated within the W3C community. URLs are just certain classes of URIs. URIs have a particular, recognizable and uniform syntax. Examples include (from [1]):

ftp://ftp.is.co.za/rfc/rfc1808.txt

http://www.math.uio.no/faq/compression-faq/part1.html

mailto:mduerst@ifi.unizh.ch

news:comp.infosystems.www.servers.unix

Note that each URI begins with a scheme, which describes a known method of access to the resource identified. Common schemes in use today are: ftp (File Transfer Protocol),
<table>
<thead>
<tr>
<th>Element</th>
<th>Attributes</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural Elements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>html</td>
<td>xmlns, xml:lang, lang</td>
<td>head, body</td>
</tr>
<tr>
<td>head</td>
<td>title, script, link, meta, style, base</td>
<td></td>
</tr>
<tr>
<td>body</td>
<td>BEs, ILEs, EOs,</td>
<td></td>
</tr>
<tr>
<td><strong>Block Elements (BEs)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td>h1, ..., h6</td>
<td>BEs, ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>BEs, ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td>blockquote</td>
<td>BEs, ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td>pre</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>br</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>hr</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ul, ol</td>
<td>li</td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>BEs, ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td>dl</td>
<td>dt, dd</td>
<td></td>
</tr>
<tr>
<td>dt, dd</td>
<td>BEs, ILEs, EOs</td>
<td></td>
</tr>
<tr>
<td><strong>Inline Logical Elements (ILEs)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abbr, acronym, cite, code, em, strong, var</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Embedded Objects (EOs)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>href, id [name]</td>
<td>EOs</td>
</tr>
<tr>
<td>img</td>
<td>alt</td>
<td>-</td>
</tr>
<tr>
<td>table</td>
<td>tr [td], th, caption, BEs, ILEs, EOs</td>
<td></td>
</tr>
</tbody>
</table>

*Markup directives are ignored within a pre block.

Table 1: XHTML tags you should be familiar with.

http (Hypertext Transfer Protocol), mailto (e-mail), news (USENET news groups), telnet (interactive services via the TELNET protocol), and file (local or remote files).

**Question 2.1**

One subset of URIs has the following basic syntax:

\[
\text{scheme}://\text{authority} \quad \text{path}?:\text{query}
\]

Describe what can appear as schemes, authorities, paths, and queries in the above class of URIs. Give an example of a URI that conforms to this syntax. Give an example of a URI that does not conform to this particular syntax.

**Tables in XHTML**

In XHTML, frames are deprecated. However, one common use of frames has been to break a web page up structurally, for example into a table of contents and a separate “content” part. Although HTML tables were originally thought of as simply units to organize data (think spreadsheets), nowadays tables are used for breaking up the structure of a web page into functional units. If you see a well-designed XHTML web page with a “contents bar” on the left, for example, that effect is likely achieved using XHTML tables. It is recommended you use tables for this purpose (as
well as data organization). For this, you need to have a reasonable facility with XHTML tables and their inherent recursive structure.

**Question 2.2**

Imitate the following table in XHTML. Use only the logical subset of HTML (in other words, do not specify the inline visual formatting directly). *Note:* you can do this by including “subtables” in larger tables. However, it is probably easier if you use the `colspan` and `rowspan` attributes appropriately.

![Table in XHTML](image)

### 2.3 CSS

The language CSS (Cascading Style Sheets) was introduced in 1996 by the W3C to bring back the separation of logical and visual markup. The idea is that an HTML (or XHTML) document contains only logical markup, and visual markup is defined completely separately in an accompanying CSS document.

The CSS language is rather simple. A CSS document consists of several *style directives* which specify how to display the contents of XHTML elements. Each style directive has the following format:

```css
/* General form of CSS directives */
[selector] {
    property1: value1;
    property2: value2;
    ...
    ...
    propertyN: valueN; 
}
```
Note that comments in CSS are begun by /* and terminated by */.
An exhaustive list of CSS properties and values can be found at

http://www.w3schools.com/css/css_reference.asp.

Helpful examples are available at

http://www.w3schools.com/css/css_examples.asp.

As an example to get you started, the following CSS directive specifies a heading of
type h1 is to be displayed as italic, 36 point sans-serif font, in the color blue:

```
    h1 {
        font-style: italic;
        font-size: 36pt;
        font-family: sans-serif;
        color: blue;
    }
```

Most XHTML elements can appear as selectors in a CSS directive, however some prop-
erties and/or values won’t make sense for every element. Your best bet is a combination
of trial-and-error, and reference to the W3C specification documents.

### 2.3.1 CSS for Visual Markup

There are three ways of applying CSS styles to an XHTML document: (1) inline, (2)
embedded, and (3) external. Inline styles use the `style` attribute of XHTML elements.
However, this method is not recommended for two reasons:

- Inline styles must be repeated wherever that style is to be applied. This involves
  unnecessary repetition and can lead to inconsistent styles within a document.
- Inline styles conflate the logical functions of XHTML with the visual functions of
  CSS.

Slightly better is to use embedded styles. An embedded style occurs in the `<head>` of
an XHTML document, in a `<style>` element. Again, however, this does not attain true
separation of logical and visual markup.

The method we will use is that of external style sheets. In this case, the CSS directives
are put into a separate file with a “.css” extension, and linked to from the XHTML
document(s) that will utilize those styles. External style sheets are linked via the `<link>`
element that occurs in the head section of an XHTML document:

```
    <link rel="stylesheet" href="mystyle.css" type="text/css" />.
```
2.3.2 User-Defined Selector Classes

Sometimes inline styles are preferred because they seem to have greater power to select particular elements (as opposed to all elements of the same type). This power is also available using external style sheets, and involves use of style classes. A style class is defined in CSS using a "classname" extension on a CSS selector. For example, we can define two types of paragraph classes as follows:

```
p.emph {
    font-style: italic;
    align: center;
}
p.norm {
    align: left;
}
/* The next class is available to any element that can have
   a font-color property. */
.red {
    font-color: red;
}
```

In an XHTML document, we can then access these styles using the class attribute:

```
<p class="emph">
This paragraph is to be emphasized; it is really important.
</p>
<p class="norm">
This paragraph is just normal.
</p>
<p class="red">
This paragraph will display as red text.
</p>
```

It is tough at first to separate the visual and functional aspects of your document so cleanly, but this is a first step toward the semantic web.

**Question 2.3**

Choose an HTML web page that you like (for example your own home page) and translate it into strict XHTML. Ensure you use only the logical subset of XHTML and include an accompanying CSS stylesheet that provides all visual formatting.

3 XML

XML addresses several goals of the World Wide Web community. HTML was also an attempt to meet these goals, but it is generally agreed that while HTML is simple and universal, it is neither powerful nor structured enough to meet the demands of the emerging semantic web.
XML is not a single language. Rather, it is a standardized set of rules for creating application-specific, extensible markup languages. Thus, XML meets the goals of providing flexible markup which organizations (or individuals) can tailor to their unique needs, while still being a standardized framework that allows Web agents to function meaningfully with each other, regardless of their differing applications. This strength is hard to ignore.

The XML standard provides for unambiguous, explicit logical markup. This is a good thing: it is generally recognized that natural languages contain far too much ambiguity, elipsis and innuendo to be of use for literal-minded, digital computers. XML also provides for error checking and document validation, like programming languages. A downside to this is that syntax matters more in XML than one might ideally like. If the computer is expecting the letter “a” and one accidentally types the letter “q”, the whole document is invalid. This is one of the perennial prices we pay for communicating with computers at their level.

XML standardization is vast and ongoing, and reading the specification itself is generally not helpful for understanding the power of XML. In this course, we will focus on a core subset of XML, which is introduced in the next sections.

Since XML provides pure logical (and semantic) markup, it is difficult to visualize its impact when faced with a simple text document that is marked up. The real power of XML lies in its complementing specifications, in particular XSLT, a language which allows you to translate structured XML documents into any other structured format (for example, XHTML). This context independence of XML documents is deeply important. The same XML document can be handled dramatically differently by different applications (this is called “repurposing”). In §3.2 we will learn a basic subset of the XSLT transformation language and use it to transform logically marked up documents to viewable XHTML. Finally, in §3.3, we will learn how to specify document models for validation.

### 3.1 Basic XML

In this section we’ll learn the rules for creating well-formed XML documents. Many of these rules will be familiar from XHTML. The main extension here is that tags can be anything we want them to be, as opposed to just the XHTML keywords. Our running example will be the XML document displayed in figure 1 (from [5]).

**Question 3.1**

Look the document in figure 1 over, from a logical markup perspective. What “real-world” scenario do you think is behind this document?

#### 3.1.1 Document Prolog

The first thing you’ll notice about an XML document is the prolog. The document identifies itself as XML. This declaration must begin any well-formed XML document. The properties of the XML declaration are optional, and there are three possibilities:

- **version** – which version of XML the document conforms to. Currently there’s only one version: 1.0.
Figure 1: Sample XML document

encoding – which encoding the document uses. Example values this property can take are ASCII, UTF-8, ISO-8859-1 and UTF-16. If this attribute is not specified, the encoding is assumed to be UTF-8.

standalone – this property comes with a value of yes or no, according to whether there are any other files which must be loaded in tandem with the current one.

Note that property values must be surrounded with quotation marks (single or double).

3.1.2 Elements

The next thing to notice about an XML document is the actual markup, which involves using elements. These are set off from the text by being enclosed in < and > signs. Opening and closing elements (closing elements are delineated by </ and >) enclose document blocks, which can contain text (character data) and other element blocks. This latter fact means that XML documents have a recursive structure, which is often visualized as a tree data structure (§3.1.4).

Elements are allowed to be empty, and there are two ways of indicating this. You can write “<myTag></myTag>” or, equivalently, “<myTag />”. As in XHTML, every element must be closed (and furthermore closed in the proper order).

Every XML document must have a single root element (also called a document element) which contains all the other elements. In our example, this is the <time-o-gram> element. Elements must be properly nested throughout the document (as in XHTML).

Tags are familiar, of course, from XHTML. However, the tag identifiers we used in figure 1 are most likely new to you. Where were these particular tags obtained from? The answer is that any legal tag identifier can be used in an XML document. Legal tag identifiers have similar rules to the names used in programming languages, namely:

- The identifier must start with a letter or underscore.

- The identifier can contain any number of letters, numbers, hyphens, periods, and underscores. (Probably more than 40 characters is unnecessarily long.) Note that
any letter in the encoding specified by your prolog is fine (so you have the whole of Unicode at your disposal).

- Stay away from reserved terms (such as xml, xmlns, or anything starting with xml).
- Stay away from colons, spaces, tabs, newlines, equals, quotation marks, and make sure you don’t use the same identifier for more than one XML element, attribute, or value.
- Identifiers in XML are case-sensitive: time-o-gram is a different identifier from Time-O-Gram or TIME-O-GRAM, for example.

Additionally, there must be no space between the opening < and the tag identifier. Extra white space elsewhere in the element is ignored.

Other than that – you are free to markup your document any way you want that conveys its logical purpose. As with programming variable names, however, it is good style to use meaningful (and suggestive) tag names.

XML is purely logical, i.e. the tags have no inherent meaning. Although “<emph>” means something (in terms of display) in XHTML, in XML it is only an identifier. It only has semantics when you give it semantics using XSLT, XML schemas, or associated techniques.

### 3.1.3 Attributes

Inside the < and > brackets, after the tag identifier, come any attributes with their associated values. Attribute names and values have the same restrictions as tag identifiers. Other restrictions and features you should know about are:

- Attributes are separated by white space.
- Attributes must have associated values and the syntax

  \[
  \text{att\_name}="\text{value}\" \text{ or att\_name}='\text{value}'
  \]

  must be used to indicate these.
- An element can only contain one occurrence of any particular attribute.
- Attributes can appear in any order. Make sure you take this into account when you are transforming your XML documents (§3.2).

One of the most fundamental XML representation issues is: when should my text unit be described by an element, and when should I use an attribute instead (or in addition)? This question depends on the design decisions made in going from the outside world and the meaning of a document into the world of markup. In the next topic, we will look at data modeling, and this type of question will plague us throughout that topic.
Structurally, an XML document can be thought of as a tree. We saw binary trees in topic I (branching factor always 2 or less), and these data structures can be generalized into trees with varying (arbitrary but finite) branching factor. Such a data structure can represent the hierarchical structure of an XML document in a convenient pictorial manner. The tree view reinforces the criterion that elements in an XML document must be properly nested, since nesting corresponds directly to depth in the tree. The tree view of our XML example from figure 1 is shown in figure 2.

Figure 2: Tree view of XML document from figure 1

Leaf nodes in an XML tree are the character data in the document. Internal nodes are labeled with tag identifiers (and attributes, if any). Note that if you read the leaves of an XML tree from left-to-right you can reconstruct the text of the original document (minus the markup). Thus, the internal nodes give the markup structure of the document, beginning with the root element (which, unsurprisingly corresponds to the root of the tree).

The tree view of an XML document will be handy to keep in mind when we see how to transform XML documents using the XSLT transformation language (§3.2).

3.1.5 Other XML Features

Two other XML features that you should know about are the following:

Comments – Comments in XML are delineated by <!-- and -->, as in XHTML.

Entity References – entity references in XML essentially function as macros. These are predefined strings standing for other strings. The most common place you’ll use these is to allow XML literals into your document as regular characters. As an example, note that < is a special character in XML. So what do you do when you want to put an ordinary < in your document, as in “x < y”? In this case, you use an entity reference to refer to the character.

Entity references begin with & and end with ;, as in XHTML. The entity reference &lt; stands for the character <. Unicode characters that your keyboard can’t create can also be entered into an XML document using entity references.
You can also predefine your own entity references in the prolog to your document, for example, the declaration

```xml
<!ENTITY me "Professor Cathy Wyss">
```

allows you to use the entity reference &me; as shorthand for the string “Professor Cathy Wyss”.

**Question 3.2**
What are the four entity references recognized in XML by default? How is a Unicode character represented as an entity reference? What is the entity reference for the character “é” in Unicode? (Give both the decimal and hexadecimal entity references for this character.)

### 3.2 XSLT

XML is a rather simple set of markup rules to learn. XML documents are “self-describing”: the document contains explicit tags indicating where particular elements fit in to the overarching logical structure. This would be a mere “curiosity”, however, if not for the associated languages which can process and structurally define XML documents.

In this section, we will learn about XSLT. XSLT is one of a family of languages derived from XSL (eXtensible Stylesheet Language). The idea is similar to XHTML: the XML gives the data structure, and the accompanying stylesheet says how to display it. But XSLT is more than simple styles; XSLT is a language for *transforming* an XML document into any other structured format. We will concentrate on producing XHTML output as the result of an XSLT transformation, however keep in mind that XSLT is far more general, and allows you to output many different formats, including tabular representations, comma-separated lists, pretty-printed documents, and more.

XSLT directives are kept in a separate file which accompanies the XML document. One nice symmetry is that an XSLT document is in fact an XML document. This is unlike XHTML and CSS, where the syntax of CSS is clearly divergent from that of XHTML.

**Question 3.3**
Can you think of any reasons why it is useful to have the XSLT syntax conform to the XML markup specification?

#### 3.2.1 General Form of XSLT Templates

An XSLT stylesheet consists of several *template* declarations. Each template picks a particular part of the XML document and describes how to transform it. The general form of an XSLT template declaration is:

```xml
<xsl:template match="[match_pattern]">

[insert character data and/or XSLT directives here]

</xsl:template>
```
In the next sections, we will describe the various parts that contribute to an XSLT template. In particular, what does a “match pattern” look like? What XSLT directives can appear in the template body, and what is their purpose? Keep these questions in mind as you read further. To understand the action of XSLT templates, it is best to conceive of an XML document in its tree form.

3.2.2 Match Patterns

A match pattern picks out particular nodes (and/or leaves) from an XML document. This idea is so important, a separate language has evolved to do this, called XPath. Every XPath expression is composed of location path steps, separated by the character “/” (forward slash). Every location path step has the following syntax:

\[
\text{axis :: node test [boolean test]}
\]

As indicated, the only part of the location path step that is mandatory is the node test. This picks out particular nodes in the tree (to which the template will be applied). The optional components help refine the search for matching nodes. The axis part of the location path step indicates a particular direction of movement within the tree. The axis specification is prepended to the node test using two colons (“::”). The boolean test gives further restricting conditions a node must satisfy to pass the match test. The boolean test (if present) occurs after the node test and is surrounded by square brackets.

**Axis Specification** As mentioned, an axis specification indicates a particular direction of movement within the tree. The choices of axis specification are: child, attribute, descendant, descendant-or-self, following, following-sibling, namespace, and preceeding-sibling. Some examples of these search axes are given in figure 3. The context node (i.e. the node that the search axis is “with respect to”) is shaded grey. Nodes circled with the dashed line indicate which nodes fall in the particular axis displayed. Note that there is a left-to-right ordering imposed on nodes in the XML tree (this order is imposed by the linear order in which the elements occur in the XML file). Note that for search purposes, text nodes (shown as boxes) and element nodes (shown as ovals) are treated the same.

Axes not appearing in figure 3 should be reasonably self-explanatory. An exception may be the attribute axis, which runs over the attributes of the context node. According to our tree notation, this means the search is internal to the context node.

Note that if an axis is not specified in the location path step, it is assumed to be child (i.e. this is the default axis).

**Node Tests** The second component of a location path step (and the only mandatory one) is the node test. This is a syntactic pattern that matches certain elements and/or text. Here are some important node tests:
(a) child axis

(b) descendant axis

(c) following axis

(d) following-sibling axis

Figure 3: XPath Search Axes in an XML Tree

/ – this matches the top of the XML document. It is as if there is an “invisible” element above the root node which gets matched. (So the root node of the XML document is a child of /.)

* – this is a wildcard. It matches any element.

x – where x is a tag identifier. This matches only elements that use that identifier.

text(y) – this matches a leaf that consists of text y.

node() – this matches any node (i.e. non-leaf).

comment() – this matches any comment.

It may be useful at this point to turn to §3.2.4 and look at the example XSLT stylesheet. Try to identify the node tests, and compare them with the resulting output.

1If you know anything about UNIX file naming, XPath is very similar.
**Boolean Tests**  In case the node test is not powerful enough to narrow the search to exactly where we want, XPath provides *Boolean tests* that can further narrow down the matching nodes. A Boolean test essentially computes a truth value. In XPath, a Boolean test can consist of a match pattern which picks out a set of nodes. If this set is empty, the value of the test is *false*; if this set picks at least one node, the value of the test is *true*.

Beyond this, the most familiar type of Boolean tests have the form `exp1 op exp2` where `op` is one of the comparison operators `=, !>, &lt;=, or &gt;=`. Note that since `<` is an XML reserved keyword, we have to use the entity reference `&lt;` to refer to the “less than” operation.

The patterns `exp1` and `exp2` can be any of the following:

- `@x` – this refers to attribute `x` of the current node,
- a string surrounded by single quotes (e.g. `'foo'`),
- a function call.

The functions you will want to know about are:

- `position()` – returns the position of the current node with respect to its siblings (leftmost is sibling 1),
- `last()` – returns true if the current node is the rightmost child of its parent,
- `count($)` – counts the items indicated by $, which is a match pattern. For example, `count(child::*)` counts the number of children of the current node.

You can link Boolean tests using the connectives `and`, `or`, and `not()` if desired.

**Shorthand Notations**  XPath can be quite “wordy”, and the following shorthands are provided for commonly used patterns.

- `@x` for `attribute::x`,
- `. for `self::*`,
- `.. for parent::*`,
- `/* for the root element of the document`,
- `// for `descendant-or-self::*`,
- `[n] for position() = n`, and
- `| to separate location path steps that give alternaties within a match pattern.`

It is not recommend you use these shorthands at first, since it will be challenging enough to learn the XSLT basics. These are provided here only so you can read *other people’s* XSLT.
3.2.3 XSLT Directives

Up to now, we have focussed on detailing the syntax for describing the set of nodes which an XSLT template will be applied to. OK, but now what types of transformations can we perform on the resulting set of matching nodes? The answer lies in the XSLT directives, which give XSLT templates their functionality.

Here are some directives you should know about:

**xsl:apply-templates** – this is a recursive call. Wherever the XSLT interpreter encounters this directive, it processes the next nodes in the node set, then returns to the directives after this one.

**xsl:value-of** – this is used to put values into the output. These can either be character data or values taken from the input document. Note that when the XSLT interpreter reaches a leaf (text) node, the text is copied into the output document (in the absence of directives specifying *not* to do this). Thus, a value-of directive is most useful for outputting calculated or variable values. The outputted value is specified using a *select pattern*.

**xsl:for-each** – this directive instructs the XSLT interpreter to loop over a node set. This node set is specified using a *select pattern*. Directives enclosed within the for-each are applied to each of the selected nodes.

**xsl:if** – this is a conditional statement which uses a Boolean test to determine whether to apply the directives enclosed within the if block. The Boolean test is specified using the `test` attribute.

**xsl:variable** – this is a variable declaration. A calculated value is chosen using a *select pattern* and assigned a name using the `name` attribute.

**xsl:sort** – this command enables you to direct the order in which the output will appear according to a *select pattern*.

To really get an idea of what each of these directives does, you should study the example in §3.2.4 carefully.

**Select Patterns** All of the directives above use *select patterns*. The idea is very similar to match patterns, and indeed, the XPath language is used for select patterns as well. A select pattern identifies a set of nodes that the directives will apply to. The only difference between select patterns and match patterns (for our purposes) is that a select pattern can pick nodes from *above* the current node as well as below. This is because a select pattern can begin with any of the match pattern axes, as well as any of the following: *ancestor, ancestor-or-self, parent, preceding*.

**Question 3.4**
What’s wrong with the following XSLT template declaration?

```xml
<xsl:template match="ancestor::*//emphasis">
...
</xsl:template>
```
3.2.4 Example

Here is an extended example of an XML document and several corresponding transformations into XHTML. The XML document in question is a personal CD library, and is shown in figure 4.

We will define three stylesheets to translate the document to XHTML, as follows:

1. The first stylesheet will simply display the text of the XML document, with a minimal amount of processing (figure 5). The combined output is shown in figure 6.

2. The next stylesheet will display the CD library as a formatted list of entries (figure 7). The combined output is shown in figure 8.

3. The final stylesheet will display the CD library as an XHTML table (figure 9). The combined output is shown in figure 10.

Note that a stylesheet is referenced in XML with a declaration such as the following in the document prolog:

```xml
<?xml-stylesheet type="text/xsl" href="[stylesheet URI]" ?>
```

This processing instruction is not shown in figure 4, but it was used to display the results below.

**Question 3.5**

Give XPath expressions to pick out the following information from the XML tree corresponding to our CD library:

1. The root node.
2. All CDs by Bryan Adams.
3. All CDs of genre “pop”.
4. All CDs that have a title and a subtitle.
5. All artists that do not have a lastname.
6. The number of artists that do not have a lastname.
<xml version="1.0" encoding="UTF-8">  
<cds>
  <cd>
    <title>The Chopin Collection</title>
    <subtitle>The Concertos</subtitle>
    <artist firstname="Artur" lastname="Rubinstein" />  
    <composer firstname="Frédéric" lastname="Chopin" />  
    <publisher>RCA</publisher>
    <genre>classical</genre>
  </cd>
  <cd>
    <title>On a Day Like Today</title>
    <artist firstname="Bryan" lastname="Adams" />  
    <publisher>A&M</publisher>
    <genre>pop</genre>
  </cd>
  <cd>
    <title>The Four Seasons</title>
    <artist firstname="Neville" lastname="Marriner" />  
    <composer firstname="Antonio" lastname="Vivaldi" />  
    <publisher>Argo</publisher>
    <genre>classical</genre>
  </cd>
  <cd>
    <title>Document</title>
    <artist firstname="R.E.M." />  
    <publisher>BMG Records, U.K.</publisher>
    <genre>pop</genre>
  </cd>
  <cd>
    <title>Pulp Fiction</title>
    <artist firstname="Various" />  
    <publisher>MCA</publisher>
    <genre>soundtrack</genre>
  </cd>
  <cd>
    <title>Dirt</title>
    <artist firstname="Alice in Chains" />  
    <publisher>Columbia</publisher>
    <genre>alternative</genre>
  </cd>
  <cd>
    <title>...Hits</title>
    <artist firstname="Phil" lastname="Collins" />  
    <publisher>Atlantic</publisher>
    <genre>pop</genre>
  </cd>
  <cd>
    <title>Adore</title>
    <artist firstname="Smashing Pumpkins" />  
    <publisher>Virgin</publisher>
    <genre>alternative</genre>
  </cd>
</cds>

Figure 4: Personal CD library in XML.
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <body>
        <h1>CDs</h1>
        <h2>Total Count: <xsl:value-of select="count(/cds/cd)" /></h2>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>

Figure 5: First Stylesheet

Figure 6: First Rendering
Figure 7: Second Stylesheet
CDs

Total Count: 8

1. The Chopin Collection: The Concertos (Artur Rubinstein) RCA
2. On a Day Like Today (Bryan Adams) A&M
3. The Four Seasons (Neville Marriner) Argo
5. Pulp Fiction (Various) MCA
6. Dirt (Alice in Chains) Columbia
7. ...Hits (Phil Collins) Atlantic
8. Adore (Smashing Pumpkins) Virgin

Figure 8: Second Rendering
Figure 9: Third Stylesheet

27
## CDs

**Total Count: 8**

### Genre: alternative

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Composer</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirt</td>
<td>Alice in Chains</td>
<td></td>
<td>Columbia</td>
</tr>
<tr>
<td>Adore</td>
<td>Smashing Pumpkins</td>
<td></td>
<td>Virgin</td>
</tr>
</tbody>
</table>

### Genre: classical

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Composer</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Chopin Collection: The Concertos</td>
<td>Artur Rubinstein</td>
<td>Frédéric Chopin</td>
<td>RCA</td>
</tr>
<tr>
<td>The Four Seasons</td>
<td>Neville Marriner</td>
<td>Antonio Vivaldi</td>
<td>Argo</td>
</tr>
</tbody>
</table>

### Genre: pop

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Composer</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>On a Day Like Today</td>
<td>Bryan Adams</td>
<td></td>
<td>A&amp;M</td>
</tr>
<tr>
<td>...Hits</td>
<td>Phil Collins</td>
<td></td>
<td>Atlantic</td>
</tr>
</tbody>
</table>

### Genre: soundtrack

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
<th>Composer</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pulp Fiction</td>
<td>Various</td>
<td></td>
<td>MCA</td>
</tr>
</tbody>
</table>

Figure 10: Third Rendering
3.3 Document Modeling

Much of XML’s promise and power lies in its flexibility. However, this flexibility also has drawbacks. For example, if both Fred and Mary store their CD collection in XML documents, nothing keeps them from using completely incompatible elements and overall document structure. Then when Fred and Mary want to access an online database of CDs (for example to receive discounts on CDs like those they own), the online database may be expecting another, entirely different structure from either of theirs.

This problem is solved by validating an XML document against a published document model or schema. For example, we validated our XHTML documents against the document model available at

http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd.

Validation always occurs with respect to a particular document model. The language of choice for expressing these document models is XML Schema. An XML Schema document model for our simple memo (figure 1) is given in figure 11, below.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <xsd:element name="time-o-gram" type="TimeOGramType" />

  <xsd:complexType name="TimeOGramType">
    <xsd:sequence>
      <xsd:element name="to" type="xsd:string" />
      <xsd:element name="subject" type="xsd:string" />
      <xsd:element name="message" type="MessageType" />
      <xsd:element name="from" type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="pri" type="xsd:string" />
  </xsd:complexType>

  <xsd:complexType name="MessageType" mixed="true">
    <xsd:all>
      <xsd:element name="villain" type="xsd:string" minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="emphasis" type="xsd:string" minOccurs="0" maxOccurs="unbounded" />
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

Figure 11: Example XSchema Definition

Details on the XML Schema language are available at [9]. Like XSLT, XML Schema produces valid XML documents. This is in opposition to the old document modeling language, DTD. DTD is a much less elegant and powerful modeling language, and has the additional drawback of not producing legitimate XML. We will not use DTDs in this course, but you can read about their syntax online [10]. Unfortunately, many published document schemas use the DTD language, although there is a movement to replace these with proper XML Schemas.
References


