

Electronic publishing with XML  
6th CEEnet workshop on networks technology  
**Budapest**

Sébastien 'sbi' BLONDEEL (<sbi@IDEALX.com>)

August 2000

**Abstract**

Networks and the speedup and ease in communications brought the possibility of interchanging documents without using paper to transmit them. XML is a new trend a few years old, and one of its aims is to facilitate establish a gap between what a document looks like, and the content it holds, (re)opening the way to the idea of writing once a document, and using several times in different formats. This slideshow uses an XML file and a Makefile transforms it to four different formats.

**Contents**

<b>1 Preamble</b>	<b>1</b>
<b>2 Pre-requisites to follow this and do the exercises</b>	<b>2</b>
<b>3 Examples of shell manipulation</b>	<b>4</b>
<b>4 Example of a Makefile</b>	<b>5</b>
<b>5 How to type characters not present on the keyboard</b>	<b>5</b>
<b>6 What is XML?</b>	<b>6</b>
<b>7 The components of XML</b>	<b>7</b>
<b>8 Special characters in XML</b>	<b>7</b>
<b>9 The XML header</b>	<b>8</b>
<b>10 Example: the header of this presentation</b>	<b>8</b>
<b>11 Syntactic rules of XML</b>	<b>8</b>

<b>12 Example: an XML file</b>	<b>9</b>
<b>13 XML compared to SGML</b>	<b>9</b>
<b>14 The goals pursued in the making of XML</b>	<b>9</b>
<b>15 Well-formedness and validity</b>	<b>11</b>
<b>16 Syntax of a DTD</b>	<b>12</b>
<b>17 Example: the DTD of this presentation</b>	<b>12</b>
<b>18 When to stop tagging?</b>	<b>13</b>
<b>19 Encoding issues</b>	<b>14</b>
<b>20 Stylesheet languages</b>	<b>15</b>
<b>21 XPath</b>	<b>18</b>
<b>22 Example: the transformation of this document</b>	<b>19</b>
<b>23 A few words about L<sup>A</sup>T<sub>E</sub>X<sub>2</sub>e</b>	<b>20</b>
<b>24 Example of a L<sup>A</sup>T<sub>E</sub>X<sub>2</sub>e document</b>	<b>21</b>
<b>25 Editing XML according to a DTD: PSGML</b>	<b>21</b>
<b>26 Assignment</b>	<b>22</b>
<b>27 Installing XML tools</b>	<b>22</b>
<b>28 DocBook</b>	<b>23</b>
<b>29 DocBook Modular StyleSheets</b>	<b>23</b>
<b>30 References</b>	<b>23</b>

## **1 Preamble**

- Electronic Publishing means using computers as sophisticated typewriters coupled with printers
- The storage format of your documents is important
- I suggest your relation to content and markup to be of a higher level than just focusing on the way things look
- Virtualizing markup helps create different versions of a document: compare them

- Semantic markup ensures more coherence than most WYISWYG environments default settings
- You are independent from software clients and interface and can profit or the same invested time to edit efficiently mail, documents, programs
- If coupled with research or database tools, semantic markup can help better qualification of information: "python" can be a snake, a man, or a programming language

Electronic publishing can mean many different things. In its simplest (and dumbest) form - which can also come out to be quite expensive - one buys some proprietary office suite and proceeds to generate various formats of files simply by using a "save as" command or some such one-click solution. However, doing this requires owning the suite and it means you depend on it from then on. Should the company disappear or simply decide to redirect its activities in this or that direction, you may end up stranded, without any recourse, and, furthermore, with very little know-how.

The approach adopted here is quite different. Based on free or open-source software, electronic publishing is viewed as a file manipulation process, that remains entirely in the hands of the users. As a result, instead of becoming passive consumers who depend on aperiodical "fix" from some software company, each one of you can not only freely produce all the documents that you wish while also contributing to the development of better tools or procedures, for example improving the support for some specific language or charset.

The approach adopted here also differs from common word processing approaches in that it rigorously separates the writing phase from the markup phase and the presentation phase. The first phase is that of creation of a document; the second one is a phase of enrichment of the document through various markup schemes; the third phase clearly focuses on the material appearance and the visual formatting of the document. Thus, the handouts you are presently reading were finally printed using  $\text{\TeX}$ . Why? Because few if any other tools can offer as good typographical control as  $\text{\TeX}$ . No word processor can reach the subtlety of  $\text{\TeX}$  for appearance. Also,  $\text{\TeX}$  allows you to manage the presentation of your document in ways that no word processor could ever even hope of approaching. At the same time, that particular issue was dealt with on its own terms and not in a mixed-bag approach where you try at the same time to handle ideas, arguments, vocabulary AND appearance in a completely inefficient and even incoherent manner.

## 2 Pre-requisites to follow this and do the exercises

- quite a transversal and complete talk
- this works in (free) unix systems, and should in others

XML tools aim to be used by everybody. They work on proprietary, technically debatable platforms. Even if you have to use them at some point, because of the cluelessness of decision-makers or your users, why use them to develop?

- know UNIX shell manipulation

See examples in next slide. Think `Makefile`, think shell function or personal quick script when some large command has to be repeated on a regular basis. Example:

```
function xt() {
    echo "XT $*" >&2
    java com.jclark.xml.sax.Driver $*
}
```

- know some window manager manipulation

Be able to have several processes running at the same time, remember where they are, and don't spend your life moving your hands around from keyboard to mouse.

This means knowing the basic commands that allow you to navigate and work effortlessly in command mode with a Unix shell. You should certainly know how to use `ls`, `cd`, `find`, etc. with all the usually useful qualifiers so as to manipulate your files quickly and easily. You should understand the pipe and symbolic link concepts. Never forget the command `man` and you might want to do `man man` at least once to get a general overview. Learn how to decipher the sometime cryptic descriptions of `man`. Before learning special concepts, learn how to walk and talk through the Unix world: `man man` will fire up a pager which you need to be able to move around, for example. I know there is some kind of a bootstrap problem here, but if there wasn't this would mean there is little new in the concepts used in Unix you don't know.

Simply put, know how to make a window active, how to move a file from a window to another or from a directory to another through windows, etc.

- know how to type, including accents not present on keyboard (ask me)

You can really save a lot of time if the keyboard is a natural extension to your body, and your eyes hardly ever quit the screen when you type. I would even suggest you (at least people from languages written with a latin-alphabet) to use an American keyboard setting and learn how to type your special characters on these. They're the most widely available everywhere, and you can always type blindly (which you should do at some point anyway) no matter what special national keyboard you have to be using

Use of codes. Remember also `man ascii`. Ask me if needed.

- know how to set up environment variables

Depending on the shell you are using (`*csh` or `*sh`), know how to temporarily (prefixing a command with the the assignment) or permanently set or change the value of a standard environment variable like `LANG`, `EDITOR`, `CLASSPATH`, ...

- know how to read, write and understand a Makefile

Try to make them roughly independent of your particular directory or account on this machine (using `ROOTDIR` and other general variables). At some point, it might be useful to spend time on the correct resolving of dependencies to save compilation time if you have to recompile often. Like in most fields, get started talking to a friend or a community or reading a technical introduction, get better reading code or the reference documentation, these things depend on your personality.

- know how to ftp and manipulate archives

Using binary mode, fetching or putting recursively, in a non interactive manner (with improved clients such as `ncftp` or `lftp`). Know how to create or extract `tar` archives compressed with either `GNU zip` (understood by standard uncompress programs in proprietary environments) or `bzip2` (more efficient, slower, available on many platforms but not installed by default on many).

You must be reasonably at ease with various compression schemes (`gzip`, `compress`, etc.) and various archival schemes, particularly `tar`. The point is that you must be able to extract archived files you receive and you must be able to archive the files you work with, for example to transmit them to someone else. For the last part, knowing the basic ftp commands is very useful (`get`, `put`, `mget`, `mput`, `bin`, `prompt`, etc.)

- never fall behind, tell me when you don't follow

This document will "real soon now" (you know what it is to think of publishing some internal, incomplete, not perfect piece of software or documentation!) be available on the [IDEALX.org](http://IDEALX.org)<sup>1</sup> website, and will get completed and improved over time. This is your unique chance to ask me in person questions if I am going too fast or am not clear enough. The only stupid question is the one that remains unasked, and chances are that many others haven't understood either, or are wondering the same thing. When I am around, I can give you personal advice and tips on how to improve your working habits (or at least suggest things you might not have thought of), much more than by mail for example.

### 3 Examples of shell manipulation

```
$ ls -al
$ apropos xml
$ man nsgmls | grep -2i variable

$ grep -2h usepackage `locate tex | \
  grep '\.tex$'` | sort -u
```

---

<sup>1</sup><http://IDEALX.org/>

```
$ for file in `locate xml`; do grep -li \  
DOCTYPE $file /dev/null; done
```

```
$ CLASSPATH=/path/to/xt/xt.jar: \  
java com.jclark.xml.sax.Driver
```

## 4 Example of a Makefile

```
ROOT_DIR=$(shell pwd)  
FILE=CEEnet2000  
XSLT=$(JAVA) com.jclark.xml.sax.Driver  
[...]  
TEXINPUTS=$(ROOT_DIR)/figure/:$(ROOT_DIR)/sty/:  
LATEX=TEXINPUTS=$(TEXINPUTS) latex \\scrollmode\\input  
  
all:    valid  
  
valid:  
        $(NSGMLS) $(FILE).xml  
[...]
```

## 5 How to type characters not present on the keyboard

- Historically: computers speak English: ASCII charset

You can see that in the `ascii` manual page. The `xfd` command, run on a simple and standard X Window System monospace (ISO-LATIN-1) font, such as 8x16, 10x20, or 12x24 according to your monitor definition, will show the same information (and more) in a manner I personally prefer:

```
$ xfd -fn 12x24
```

- Address space too small (1 octet = 256 possibilities).

It is always very difficult to change things once they have settled, even if there are very bad and the new standard is much better. Think IPv4.

- Different pages: ISO-LATIN-1, ISO-LATIN-2, ...

They use the second half of the character set for special characters with diacritical marks and the like. There are many more ISO-LATIN-1 fonts than ISO-LATIN-2 for example.

To find or list and then view fonts available on your system:

```
$ locate fonts | grep -i cyr
$ cd <directory>
$ zcat <font.ext.gz> | strings | less
```

- Trick: commands

- setfont, setmetamode (in console)

`setfont`: choose another font in console mode. Choose one with the extended characters you need. Depending on BIOSes and hardware, the right one may be different.

`setmetamode`: bind the *Alt* key to behaving like a *Meta* modifier, not *Escape*. I personally prefer that to using dead keys or *Compose*, but your mileage may vary. Of course you have to learn them by heart but there's not that much to learn, you do it little by little, there is logic in it, and I'd rather learn a little more to be more efficient. For example, in ISO-LATIN-1, *Alt-i* will produce the character number  $105 (i) + 128 = 233$  (é). And it is most of the time upper-case coherent.

These things are in my `.profile`.

- setxkbmap (under X Window System)

This command used to be longer to remember, using `xmodmap` and a filename depending on the system configuration. Fortunately, this higher level command appeared, and the completion works after only a number of characters which are on the same place in French and American keyboards; I don't know about your national keyboard maps...

If you use the X Window System, I suggest you do most of the work in big font terminals, and you use `xterm`, since most other terminals have a problem or some limitation. Beware that the default settings of some window managers intercept the events corresponding to the production of some accented characters and will iconify a window for example. Graphical browsers I know do not recognize them either. *Emacs* must be configured. *Vim* works fine by default, if called under that name.

- How to type Chinese, Japanese? Ex: `cxterm`

This program is hard to compile but many interesting encoding possibilities. You may type ASCII or ideograms, and type ideograms by English name, traditional Chinese pronunciation (with or without the tones), or Cantonese pronunciation. If a character is not ASCII, then the following one is read together with it so as to form an ideogram. The Chinese fonts are twice as wide as Latin fonts to ensure alignment in mixed texts.

- Unicode coming slowly.

This might solve compatibility and exchange of documents, but we will probably still use a limited charset locally and only use Unicode to exchange stuff, since it eats up more space and most of the programs are not really designed to support this in input.

## 6 What is XML?

The eXtensible Markup Language<sup>2</sup> is a few years old.

It is being normalized by the W3C: drafts and recommendations.

Standards are still being worked on.

W3C is still producing recommendations and drafts, and things are still hot and boiling, changing rapidly. However, some things already work, but many things have not yet settled down, and the software available is either alpha, or incompatible with the examples shipped with another piece of software.

It is a meta-language.

It is just syntactic sugar.

When I say that XML is syntactic sugar, I mean that XML, while exceedingly useful, does not solve all the problems under the sun. In particular, it does not solve semantic problems by itself, even though it can be used to enrich the semantic value of a document.

You might never have learned to program in assembly language, or with punched cards. Now these had really annoying and limiting syntax to work with. Do you really believe the higher level things that have popped up since were a silver bullet as far as software engineering was concerned? An easy syntax can let the scarce resource: the human brain and attention, focus on the interesting things rather than spend one's time on boring repetitive details machines are much better at.

## 7 The components of XML

- a header
- tags surrounding elements
- attributes
- entities
- comments
- text content
- CDATA sections
- and some more (directives, ...)

---

<sup>2</sup><http://w3c.org/XML/>

## 8 Special characters in XML

Because XML has a syntax similar to that of HTML, some characters may not be reproduced literally. These are:

- left angle bracket (actually, "less than"): &lt;
- right angle bracket ("greater than"): &gt;
- ampersand: &amp;
- single quote (or "apostrophe"): &apos;
- double quote: &quot;

## 9 The XML header

Must specify:

- the version of XML (mandatory)
- the encoding used (if non-ASCII characters)

Can specify:

- other attributes (stand-alone, ...)
- a DTD, with a PUBLIC and SYSTEM identifier
- parameter entities to ignore parts (like draft) or include the contents of a file

## 10 Example: the header of this presentation

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE presentation PUBLIC
"-//Sebastien Blondeel//DTD presentation slides//EN"
"dtd/slides.dtd" [
  <!-- extra entities -->
]>
<presentation>
  <title>Electronic publishing with XML</title>
  <conference>6th CEEnet workshop on networks technology</conference>
  <place>Budapest</place>
  ...
```

## 11 Syntactic rules of XML

- Size (of letters) does matter: tags are case sensitive
- Some characters are forbidden in tag names
- Tags must be closed
- Special case: empty tags
- Elements must be properly nested
- Attributes must be quoted (or double-quoted)

## 12 Example: an XML file

```
<?xml version='1.0'?>
<root>
  <foo lang="hu">
    <anchor id="fooAnchor">
      Friend or foo?
    <bar>I started after, I must finish first.
    </bar> <!-- no attributes in closing tags -->
  </foo>
  <blah>
    Another element directly under the root element.
  </blah>
</root>
```

## 13 XML compared to SGML

The Standard Generalized Markup Language is older and much more complicated:

- tag delimiters can be redefined
- opening or closing tags can be omitted
- empty tags can be implied
- some elements can be empty if and only if some attribute has a special value

## 14 The goals pursued in the making of XML

(courtesy of "A Technical Introduction to XML"<sup>3</sup>, by Norman Walsh)

The XML specification sets out the following goals for XML:

- straightforward to use over the Internet (when on the fly XML-aware browsers exist and work)

It shall be straightforward to use XML over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents. In practice, this will only be possible when XML browsers are as robust and widely available as HTML browsers, but the principle remains.

- shall support many applications (more than just web content: authoring, content analysis, ...)

XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc. Although the initial focus is on serving structured documents over the web, it is not meant to narrowly define XML.

- easy to parse: two weeks for a competent graduate student

XML shall be compatible with SGML. Most of the people involved in the XML effort come from organizations that have a large, in some cases staggering, amount of material in SGML. XML was designed pragmatically, to be compatible with existing standards while solving the relatively new problem of sending richly structured documents over the web.

- compatible with SGML for backward-compatibility reasons

It shall be easy to write programs that process XML documents. The colloquial way of expressing this goal while the spec was being developed was that it ought to take about two weeks for a competent computer science graduate student to build a program that can process XML documents.

- as few optional features as possible (for portability)

The number of optional features in XML is to be kept to an absolute minimum, ideally zero. Optional features inevitably raise compatibility problems when users want to share documents and sometimes lead to confusion and frustration.

- human-readable and clear

XML documents should be human-legible and reasonably clear. If you don't have an XML browser and you've received a hunk of XML from somewhere, you ought to be able to look at it in your favorite text editor and actually figure out what the content means.

---

<sup>3</sup><http://www.xml.com/pub/98/10/guide1.html>

- design must be achieved quickly

The XML design should be prepared quickly. Standards efforts are notoriously slow. XML was needed immediately and was developed as quickly as possible.

- design must be clear and concise (EBNF)

The design of XML shall be formal and concise. In many ways a corollary to rule 4, it essentially means that XML must be expressed in EBNF and must be amenable to modern compiler tools and techniques.

There are a number of technical reasons why the SGML grammar cannot be expressed in EBNF (Extended Backus-Naur Form)<sup>4</sup>. Writing a proper SGML parser requires handling a variety of rarely used and difficult to parse language features. XML does not.

- documents easy to create: text editors, scripts...

XML documents shall be easy to create. Although there will eventually be sophisticated editors to create and edit XML content, they won't appear immediately. In the interim, it must be possible to create XML documents in other ways: directly in a text editor, with simple shell and Perl scripts, etc.

- terseness does not matter: compressing software exists, and saving typing complicates the design and the parsing

Terseness in XML markup is of minimal importance. Several SGML language features were designed to minimize the amount of typing required to manually key in SGML documents. These features are not supported in XML. From an abstract point of view, these documents are indistinguishable from their more fully specified forms, but supporting these features adds a considerable burden to the SGML parser (or the person writing it, anyway). In addition, most modern editors offer better facilities to define shortcuts when entering text.

## 15 Well-formedness and validity

When an XML document satisfies the syntactic rules cited above, then it is called *well-formed*.

The whole point of XML is to give semantics to content. There are ways to describe the structure of the tags and elements an XML document must follow. This is being done using a Document Type Definition, or DTD.

DTDs are very poor schemes. They are not XML documents, therefore cannot be easily checked with usual XML tools.

DTDs have no typing and very little expressivity. Most DTDs published are full of mistakes and cannot possibly be used by those who publish them as "examples" on their websites.

---

<sup>4</sup><http://www.xml.com/pub/98/10/guide5.html>

When the structure of a document obeys the rules given in the DTD that accompanies it, then this document is called *valid*.

"Typing" here means ensuring or imposing a variable to belong to a certain set, or be interpreted as an integer, a string, or else. One may wish to define his own types in the rules defining the structure of an XML document, so as to ensure this is an internal identifier, this is a date in the ISO format, this is...

DTDs cannot count very well. To them, there are the numbers nothing, zero or one, zero or more, one or more. One may wish to be able to define finer intervals without piling up jokers or the like.

Sometimes, one may need or wish to have a certain field belong to a fixed list of values, with a special case for "Other", which would come with a value not in the list. DTDs won't let you do that unless you don't mind wasting elements and complicating the structure of the document.

DTDs exist for historical reasons. Efforts are being made to replace them with something better. XML-Schemas are a good candidate.

But DTDs are what now exists and is supported, and it's not that long to learn their syntax so let's start there.

## 16 Syntax of a DTD

A DTD defines the structure of a tree. For each element of the tree, it will give the possible sons it may have, in what order, and in what number.

The root element used for the element is not given in the DTD but in the header of the document.

A DTD may be referred to, in the header of the document, by a *SYSTEM* identifier, or by a *PUBLIC* identifier, which must then be defined in one of the catalog files used (either in command line or in the appropriate environment variable).

Choosing between attributes and subelements is not always easy. Choosing to use layers of wrapping elements or to use a flatter organization may as well influence maintenance and development. Big DTDs use entities and families of elements.

There are two types of elements: block elements, and inline elements.

Block elements are paragraphs, titles, tables, nested itemized lists. Inline elements just happen in the normal flow of text inside a block element and usually apply to one or a few words, inducing a local change of font. They do not really qualify the main structure of a document, but give more semantics to words or phrases.

## 17 Example: the DTD of this presentation

```
<!ENTITY % my-entities SYSTEM "../entities/all_entities">
%my-entities;
```

```
<!ELEMENT presentation (title, conference, place, date, au-
thor,
```

```

                                abstract, slide+)>
<!ELEMENT title                (#PCDATA|emph|kbd|code|strong|file|ulink)*>

<!ELEMENT author                (firstname, nickname?, middlename?, last-
name, email?)>
<!ELEMENT firstname            (#PCDATA)>

<!ELEMENT abstract              (#PCDATA|emph|kbd|code|strong|file|ulink)*>
<!ELEMENT slide                 (title,content)>
<!ELEMENT content               ((para|itemizedlist|screen|more)+)>

<!ELEMENT ulink                 (#PCDATA|emph|kbd|code|strong|file|ulink)*>
<!ATTLIST ulink                 url CDATA #IMPLIED>

```

## 18 When to stop tagging?

- Defining inline tags is important: all your needs should be covered.
- The library of Congress distinguishes between keywords and subjects

There is an incompatibility between letting more freedom to people to describe their documents, and having a limited or fixed set of keywords to be able to do efficient research without having to think of, and use, different synonyms.

Subjects are constrained to limited lists of values, while keywords are free. This idea got copied in DocBook (one of the most important technically oriented DTDs around).

- the more you tag, the longer it will take to produce your document. If you use many different inline documents, changes will not be noticeable in the fonts used.
- DTDs and stylesheets are interesting if used in a coherent family of documents, like in a website. The more inline elements, the less chances are you will remain coherent, yielding to false negatives.
- However, a minimal tagging may be useful (see personal anecdote in full version)

I was working on the typesetting and presentation of the user's manual of some program. The document was written in French and then passed to professional translators for an English version.

This user guide was talking about buttons and menu entries, and used an italic font to display them (to show they were not ordinary words but things to look for in the GUI). The first problem was that, working in a WYSIWYG interface, the author had at times used an italic font, and sometimes a slanted font. He had used an italic font for several other things too, like company names or program names.

The morning of the day we're supposed to hand the printed version out to the printer, we notice a terrible thing: the professional translators had guessed the translations of the menu and button names. They did not match the localization of the software, so they didn't match the screenshots. It was impossible to hand that out.

So, I asked the author, while I was hacking on the index, to check out all these things and replace these poorly guessed translations with the correct name, as seen from the screenshot (since we had just received the pictures).

There were many pages to check, but he came back to me after only a little while, telling me he was finished, and yes he was sure to have forgotten nothing. "How can that be?" did I ask.

"Well, I was happy to notice I just had to look for the tags 'menuentry' and 'menubutton', and when I found one, check out the screenshot that was next to that paragraph of text".

I had spent quite a while replacing all physical calls to italic or slanted fonts with an intermediate, virtualized layer, defining these new tags exactly as italic or slanted or bold fonts. We were receiving the benefits of that policy.

## 19 Encoding issues

The standard, default encoding for XML and XML tools is UTF-8<sup>5</sup>, one of the Unicode transformation formats. UTF-8 works as follows:

Unicode transformation formats are used to represent more than 256 characters with 8 bits.

The binary representation of the character's integer value is thus simply spread across the bytes and the number of high bits set in the lead byte announces the number of bytes in the multibyte sequence:

bytes	bits	representation
1	7	0vvvvvvv
2	11	110vvvvv 10vvvvvv
3	16	1110vvvv 10vvvvvv 10vvvvvv
4	21	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv

It is not always possible to force standard XML tools to use such or such an encoding in their output; most will use UTF-8. Therefore, if you want to use a backend with no UTF-8 compatibility (or you don't know how to use it :)), then you have to include some filter pipe at some point in your `Makefile` to take the produced content back to whatever encoding you wanted in the first place. This is what the present document compilation chain does.

In C, the algorithm from Latin-1 goes as follows (and the algorithm from another encoding probably looks very similar, changing the value of the constants):

---

<sup>5</sup><http://czyborra.com/utf/>

```

putwchar(c)
{
    if (c < 0x80) {
        putchar (c);
    }
    else if (c < 0x800) {
        putchar (0xC0 | c>>6);
        putchar (0x80 | c & 0x3F);
    }
    else if (c < 0x10000) {
        putchar (0xE0 | c>>12);
        putchar (0x80 | c>>6 & 0x3F);
        putchar (0x80 | c & 0x3F);
    }
    else if (c < 0x200000) {
        putchar (0xF0 | c>>18);
        putchar (0x80 | c>>12 & 0x3F);
        putchar (0x80 | c>>6 & 0x3F);
        putchar (0x80 | c & 0x3F);
    }
}

```

In Perl, the algorithm goes as follows:

```

#!/usr/local/bin/perl -p
# assemble from iso-latin-1 to UTF-8
sub utf8 { local($_)=@_;
    return $_ < 0x80 ? chr($_) :
        $_ < 0x800 ? chr($_>>6&0x3F|0xC0) .
            chr($_&0x3F|0x80) :
            chr($_>>12&0x0F|0xE0) .
            chr($_>>6&0x3F|0x80) .
            chr($_\textampersand{ }0x3F|0x80);
} s/<U([0-9A-F]{4})>/&utf8(hex($1))/ge;

```

The reverse algorithm, leading from UTF-8 to ISO-LATIN-1, is a bit more complicated:

```

#!/usr/local/bin/perl -p
# disassemble non-ASCII codes from UTF-8 stream
$formatt=$ENV{"UCFORMAT"}||'<U%04X>';
s/([\xC0-\xDF])([\x80-\xBF])/sprintf($formatt,
unpack("c", $1)<<6&0x07C0|unpack("c", $2)&0x003F)/ge;
s/([\xE0-\xEF])([\x80-\xBF])([\x80-\xBF])/sprintf($formatt,
unpack("c", $1)<<12&0xF000|unpack("c", $2)<<6&0x0FC0|unpack("c", $3)&0x003F)/ge;
s/([\xF0-\xF7])([\x80-\xBF])([\x80-\xBF])([\x80-\xBF])/sprintf($formatt,

```

```
unpack("c", $1) << 18&0x1C0000 | unpack("c", $2) << 12&0x3F000 |  
unpack("c", $3) << 6&0x0FC0 | unpack("c", $4) &0x003F) /ge;
```

Future versions of Perl will provide a module for this. The web document quoted also gives a number of solutions or pointers for other programming languages.

## 20 Stylesheet languages

A stylesheet transforms an XML document according to rules and templates applied recursively, starting at the root element. The document needs to be well-formed, not necessarily well-formed.

DSSSL<sup>6</sup> (*Document Style Semantics and Specification Language*), is a stylesheet language similar to Scheme, embedded in XML documents.

A full DSSSL stylesheet with block elements, inline elements, and inheriting parameters (courtesy of DocBook: the Definitive Guide<sup>7</sup>).

```
<!DOCTYPE style-sheet PUBLIC "-//James Clark//DTD DSSSL Style Sheet//EN">  
  
<style-sheet>  
<style-specification>  
<style-specification-body>  
  
  (element chapter  
    (make simple-page-sequence  
      top-margin: 1in  
      bottom-margin: 1in  
      left-margin: 1in  
      right-margin: 1in  
      font-size: 12pt  
      line-spacing: 14pt  
      min-leading: 0pt  
      (process-children)))  
  
  (element title  
    (make paragraph  
      font-weight: 'bold  
      font-size: 18pt  
      (process-children)))  
  
  (element para  
    (make paragraph  
      space-before: 8pt  
      (process-children)))
```

---

<sup>6</sup><http://www.jclark.com/dsssl/>

<sup>7</sup><http://oasis-open.org/docbook/documentation/>

```

(element emphasis
  (if (equal? (attribute-string "role") "strong")
    (make sequence
      font-weight: 'bold
      (process-children))
    (make sequence
      font-posture: 'italic
      (process-children))))

(element (emphasis emphasis)
  (make sequence
    font-posture: 'upright
    (process-children)))

(define (super-sub-script plus-or-minus
        #!optional (sosofo (process-children)))
  (make sequence
    font-size: (* (inherited-font-size) 0.8)
    position-point-shift: (plus-or-minus (* (inherited-font-size) 0.4))
    sosofo))

(element superscript (super-sub-script +))
(element subscript (super-sub-script -))

</style-specification-body>
</style-specification>
</style-sheet>

```

XSL<sup>8</sup> (*eXtensible Style Language*) is a recent recommendation of the W3C. The stylesheets are XML documents. This is the stylesheet language used for the various versions of this presentation.

The language uses two *namespaces*: the `<xsl:` and the `<fo:`.

The first one, *XSL Transformations* or XSLT, is specialized in the recursive transformation of XML trees into usually other trees (XML or XHTML: because the HTML tags appear in the stylesheet between XSL-T tags, all must be closed and properly nested or else the XSLT would not be a well-formed XML document. Therefore the HTML produced must be XHTML).

The second part, *Formatting Objects* or FO, is an XML vocabulary for specifying formatting semantics. Unfortunately, this directly produces PDF in a way that can hardly be better than using directly plain TeX or L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub>, and nearly no tool works properly and recognizes the whole current state of the draft. They all are incompatible, limited to small documents when they use some flavour of TeX because they pile up useless nested environments quickly filling up the memory, and can hardly work with their own examples, and not with the examples shipped with other alpha- or beta- applications.

Example of an XSL stylesheet (using FO):

---

<sup>8</sup><http://www.w3.org/Style/XSL/>

```

<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="para">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="emphasis">
  <fo:sequence font-style="italic">
    <xsl:apply-templates/>
  </fo:sequence>
</xsl:template>

<xsl:template match="emphasis/emphasis">
  <fo:sequence font-style="upright">
    <xsl:apply-templates/>
  </fo:sequence>
</xsl:template>

</xsl:stylesheet>

```

## 21 XPath

XPath<sup>9</sup> is the W3C recommendation for expressing paths and referring to nodes in an XML tree.

It is used in XSLT to select recursively some descendants or brothers of the node currently being worked on. There is a good piece of documentation<sup>10</sup> under the XSLT tool SAXON, even if I find that tool a bit buggy and I prefer using XT.

The SAXON<sup>11</sup> documentation for XPath explains more clearly than the normative reference, a number of things:

- Constants, Variable References
  - string constants are written between single or double quotes
  - numeric constants are written using the Java rules
  - there are no boolean constants; one should use the true() and false() function calls instead

---

<sup>9</sup><http://www.w3.org/TR/xpath>

<sup>10</sup><http://users.iclway.co.uk/mhkay/saxon/expressions.html>

<sup>11</sup><http://users.iclway.co.uk/mhkay/saxon/>

- the value of a variable can be obtained using its name preceded with a dollar sign: \$name; the variable must have been declared first
- Parentheses and operator precedence
  - if an expression is enclosed in parentheses it takes precedence
  - else, operator precedence is as follows: predicate, child or descendant nodes, union, arithmetic multiplicative operations, arithmetic additive operations, comparisons, test of equality, boolean and, boolean or
- String Expressions
 

There are a number of functions operating on strings, letting one transform an expression in a string, normalize the blanks in a string, translate some characters in other characters, testing whether something is a substring of a given string, and one can even generating an id for an element on the fly, this id being compatible with the order of the elements and unique in the session.
- Numeric Expressions
 

One can obtain the position of node among a node list (this is how I numbered the slides and the table of contents in the HTML versions of this document), one can transform an expression in a number obeying certain formatting rules, ...
- Boolean Expressions
 

The number 0, the empty string, the empty node are treated as false, everything else is treated as true (this is not the case in DSSSL, which can lead to confusion). One can perform classical boolean tests and check whether a given node has a language attribute (inherited or specified).
- Nodeset Expressions
 

One can specify relative or absolute paths, use wildcards, refer the the parent node, the preceding-sibling node, the attributes named this or that in the current node or one of its descendant nodes, select all descendant nodes having such a parent and such an attribute, ...

## 22 Example: the transformation of this document

```
<xsl:template match="/presentation">
  <html><head><title>
    <xsl:apply-templates select="title"/>
  </title></head>
  <body>
    <xsl:call-template name="introduction"/>
    <xsl:call-template name="toc"/>
  </body>
</template>
```

```

    <xsl:apply-templates select="slide" />
  </body>
</html>
</xsl:template>

```

One can call a template by name, to make the stylesheet more modular and readable. One can as well apply templates recursively, selecting templates matching an XPath expression or not. XSLT is a functional language designed to be easy to implement, but the drawback is it is very verbose and one cannot reffect variables. This is particularly a problem when one needs to transform XML-defined tables into L<sup>A</sup>T<sub>E</sub>X<sub>2</sub> $\epsilon$ , where the syntax is extremely different (as opposed to what happens with the HTML syntax for tables).

This is an example of an IF-THEN-ELSE statement in XSLT (excerpt from the stylesheets of this presentation):

```

<xsl:variable name="prevslide">
  <xsl:choose>
    <xsl:when test="$slidenumber > 1">
      <xsl:number value="$slidenumber - 1" format="01" />
      <xsl:text>.html</xsl:text>
    </xsl:when>
    <xsl:otherwise>index.html</xsl:otherwise>
  </xsl:choose>
</xsl:variable>

```

Imagine what this will start looking like if you have to nest several such tests! I wrote in XSLT a function to return the age of someone knowing today's date and that person's birthday. One first has to compare the year numbers, then if they are equal the months numbers, then the day in the month. It quickly took a whole screen!

Another problem is that whitespace is handled in a mysterious way to may in XSLT. In languages where whitespaces are not very significant, such as HTML, this is not a problem, but in languages designed in a very different way, such as L<sup>A</sup>T<sub>E</sub>X<sub>2</sub> $\epsilon$ , this can be a problem. It is very hard to have a properly formatted, easy to read, stylesheet, while still producing a properly formatted, easy to read, and without bugs, generated document.

## 23 A few words about L<sup>A</sup>T<sub>E</sub>X<sub>2</sub> $\epsilon$

Donald Knuth invented T<sub>E</sub>X and then Leslie Lamport added a higher level layer automating things and giving stylesheets, a package named L<sup>A</sup>T<sub>E</sub>X, which then got further improved and named L<sup>A</sup>T<sub>E</sub>X<sub>2</sub> $\epsilon$ .

It was not meant to be used with XML and XSLT, but it is the best tool I know to produce correct printable documents. Its balancing algorithm, for example, works at a paragraph level with penalties, ensuring a better look to the document. By default,

the user has little parameters to act upon and little freedom to change the presentation and the formatting of the document, which is a good thing since styles were written by people who know what they were doing.

It has a different syntax, not using angle brackets but braces for local changes (the equivalent of inline elements), and `begin` and `end` keywords for environments. The backslash character introduces the commands, and the command name stops at the first non-letter character met (roughly).

But it was many special characters incompatible with those of XML, blank lines are significant, it has many packages with some subtle incompatibilities (for example: the footnote calls inside tables don't work), and I am not sure it can handle UTF-8 encoding, which explains the filter pass I added in the Makefile to handle this presentation (although there are a number of styles and encodings supported; I managed this week to produce Georgian and Russian documents with it, and you will be able to see on the *Not so short introduction* that it can handle a number of documents.

It is possible to draw one's fonts with the `metafont` program.

Documentation:

- Comprehensive TeX archive network.
- Not so short introduction to L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub>, which exists in 10 languages under CTAN:/info/lshort/.
- tetex documentation

## 24 Example of a L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> document

```
\documentclass{article}
% \usepackage[...] % languages, encodings
\title{The title of my document}
\author{John Doe}
\begin{document}
\maketitle
\begin{abstract}
  This document aims at giving a simple
  example of a \LaTeX2ε{} document. Understanding
  this language is \emph{interesting}.
\end{abstract}

\section{Introduction}
...
\end{document}
```

## 25 Editing XML according to a DTD: PSGML

PSGML<sup>12</sup> is an Emacs mode to assist the editing of an XML or SGML file according to a given DTD (specified with a system identifier or a public identifier).

It is triggered by the extension of the file being edited (`xml`) or (`sgml`) and these are the important keystrokes:

- C-c C-p loads (and checks) the DTD as specified in the header
- C-c C-e insert an element where the cursor is, asking you which one if the DTD gives a choice. When an element is chosen and it has implied attributes, PSGML will prompt you for them. It will also recursively fill in all it is sure about for the descendants of that element
- C-c C-v validates the document against the DTD (using `nsglms`).

I am not sure something similar exists for `vi`-clones users, I personally know the DTDs I use, or check the documentation.

If that package is not included by default in your GNU/Linux distribution, then you need to get the source code and install it, compiling and putting the `elc` files in the right place. There is a French article<sup>13</sup> detailing the steps of this installation process. It focuses on the (deprecated) LinuxDoc DTD but can be used with other DTDs.

## 26 Assignment

Now that you know a bit about XML, DTDs, XSLT,  $\LaTeX$ 2e, and you have references and pointers to the documentation, this is an exercise for you:

- Write a DTD with an element `person` describing the various things an association or company need to know about a person: name, birthdate, salary, phone number, ...
- Write a phony document with a number of people obeying your DTD. You can take the names of the students in the group or anything else.
- Write an HTML stylesheet presenting that list of people in a list, in a table, computing automatically their age (today's date being passed as a parameter to XSLT), sorting them according to their family name, salary, then age.
- Write a  $\LaTeX$ 2e stylesheet to have a nice printable version of that list

My advice: go little by little, start with a small DTD to be able to handle it, and add elements and features one by one

---

<sup>12</sup><http://www.lysator.liu.se/projects/>

<sup>13</sup><http://www.linux-france.org/article/appli/emacs/>

## 27 Installing XML tools

Most tools (above all XSLT tools) are written in JAVA so as to ensure the maximum portability. A JAVA package is a `.jar` file, actually in the ZIP format, holding a whole tree structure with classes, that must be present in the `CLASSPATH` environment variable to be found. This is the case for XT, that needs XP, both can be downloaded from James Clark's website<sup>14</sup>.

Python and Perl tools are still being worked on. The important Python XML suite of the moment is 4Suite<sup>15</sup>. I am not quite sure whether a complete Perl suite exists, since last time I checked it was rather incomplete.

The environment variable specifying the location of catalog files must be properly set if you wish to use public identifiers.

If you wish to use  $\TeX$ <sup>16</sup> as an intermediate language to produce printable versions of your document, you need to install a complete  $\TeX$  distribution, which is not an easy thing to do. I suggest you use the `tetex` distribution, that comes at least with GNU/Linux distributions.

## 28 DocBook

One of the only technical DTDs available.

Pretty complete: hundreds of elements and entities.

Hard to manage and remember; PSGML can help, or else use the reference documentation or start with a document similar to the one you wish to type.

Only one set of stylesheets available, and not working very well, especially the printed version.

DocBook.org<sup>17</sup> and Oasis-open.org<sup>18</sup>.

## 29 DocBook Modular StyleSheets

Use `Jade` to make all the transformations and `Jadetex` for the printable versions.

Mainly written in DSSSL and aimed at SGML processing (but can manipulate XML documents as well, even if the converse is not true). The XSL version supports less backends.

Can be found on Norman Walsh's website<sup>19</sup>.

May be customized<sup>20</sup>.

Assignment:

- Install DocBook, Jade, JadeTeX

---

<sup>14</sup><http://www.jclark.com/>

<sup>15</sup><http://fourthought.com/4Suite/>

<sup>16</sup><http://www.tug.org/>

<sup>17</sup><http://www.docbook.org/>

<sup>18</sup><http://oasis-open.org/>

<sup>19</sup><http://www.nwalsh.com/docbook/dsssl/>

<sup>20</sup><http://www.nwalsh.com/docbook/dsssl/doc/custom/>

- Download the source for the reference documentation for DocBook
- Cut a little bit of it and compile it into different formats: HTML (one or multi), RTF,...
- Write your own DocBook document: a short introductory paragraph then a table with the schedule of the week. Compile it to different formats.

## **30 References**

- XML on W3C
- James Clark: XP, XT
- Norman Walsh: website using XML, stylesheets for DocBook, presentations
- DocBook: a complete technical SGML/XML DTD
- T<sub>E</sub>X Users Group
- Comprehensive T<sub>E</sub>X Archive Network