

Concept of XML
7th CEEnet workshop on networks technology
Budapest, Hungary

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

August 2001

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. You will be given a pointer to all the files and compilation procedures used to produce this presentation.

Contents

1 Who am I?	2
2 Preamble	2
3 Pre-requisites to follow this and do the exercises	5
4 Examples of Unix shell manipulation	7
5 Example of a Makefile	8
6 How to type characters not present on the keyboard	9
7 What is XML?	10
8 The components of XML	11
9 Special characters in XML	12
10 The XML header	12

11 Example: the header of this presentation	12
12 Syntactic rules of XML	13
13 Example: an XML file	13
14 XML compared to SGML	13
15 The goals pursued in the making of XML	13
16 Well-formedness and validity	15
17 Syntax of a DTD	16
18 Example: the DTD of this presentation	16
19 When to stop tagging?	17
20 Encoding issues	18
21 Lab on Concept of XML	20

1 Who am I?

- Picture of the place where I was met
- Personal interests and websites
- What I may famous for, or proud (?) of

Most of you know me from last year or face to face conversations this year. For linguistic (sometimes shortcut if we can use another common language than English) and cultural reasons, we may have a tinier bandwidth than expected: the lecturer to attendee relation is probably much different in Western and Eastern countries, and it takes someone that knows both systems well to explain the difference.

A search on the Internet (nobody no longer has secrets these days) should provide you with a quick profile of me. In a few words, I work in Paris in a services company (SSII in French) where I am mostly responsible for document-handling and workflow activities as a project manager. On a personal point of view, I am involved with the Free Software movement (which will probably show at some points of this talk), translate texts, technical or not, from English to French mainly, and try to build and maintain a worldwide network of friends.

2 Preamble

- XML is a new, poorly understood, and over-hyped concept.
- It is hardly anything more than syntactical sugar

- One should distinguish well between XML as a communications media and data storage, and XML as a way to write documents
- 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 WYSIWYG 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

There are many new applications and websites about XML. It is more and more being talked about, like the new fashion everyone has to know about and everyone talks about without ever knowing it well. I remember from the first day that some of you had specific questions about XML. On my last lecture spot, I am willing to address specific questions and perform demonstrations. I do not know the answers to all questions. Some questions may mean little or nothing at all in relation to XML. I will probably not be able to find the answers to all questions by then. However, I encourage you to come and talk to me and meet me and tell me about your wills and wishes, and I will see what can be done, what I can find about them. If I do not have enough time to find good answers to your questions I will continue investigating and you can check the final version on my paper on the CEENet website and keep in touch with me by mail.

No matter what the fuss is, the core concept is very simple, and if I dared exaggerating a bit I would say it can compare to agreeing on a common alphabet: would Russian read any easier to people not speaking Russian if it switched from using Cyrillic to using Latin letters? I do not think so. However, such a move (which I do not wish and only mention as an experiment of thought) would make things easier for other people to read the words. There would no longer be hesitations as to how to read such or such letter. On the other hand, difficult and interesting problems would remain the same, unsolved, and people could concentrate and save their brains to work on them. You may be used to problems of character encodings between languages and programs. These problems are boring. One can train and be very efficient in solving them, but he would end up knowing little more than hot plate answers to a given set of questions, and may prove unable to come up with a cunning idea on a new issue. XML is little more than agreeing on a common denominator, character set (or character sets), a bit like choosing between the decimal point or the comma, or between square and curly brackets.

XML is used in many different types of applications. I will distinguish between two main types of applications: the storage and transmission of data, and the well intended but probably often self-deceiving will to come up with a document format independent of the background. I will call these two approaches "data (storage and transmission)" and "written documents".

The choice of XML as a common format (taking into account what I mentioned earlier) can be compared to the fact I am now using English to speak to you. It is the simple fact that for "large" (starting at, say, 3) values of n , n square is much more than $2n$. It means than using a common pivot language or format is simpler and faster than for everyone of us to learn every other's language. XML is just a bus between two applications, which both need to come up with input and output translation schemes. As a matter of fact, my company is working on such a thing now. The obvious drawback is the fact that it takes two translations for the idea in my brain to end up as an idea in your brain, and this increases the risks of problems in communications.

I try myself to use it as little as possible, but the little I know about what most people do is the following: Electronic handling of documents 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 with XML, in its pure form, is quite different. Based on free or open-source software (having at least such implementations), using open and documented standards, 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 $\text{T}_{\text{E}}\text{X}$. Why? Because few if any other tools can offer as good typographical control as $\text{T}_{\text{E}}\text{X}$. No word processor can reach the subtlety of $\text{T}_{\text{E}}\text{X}$ for appearance. Also, $\text{T}_{\text{E}}\text{X}$ 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.

3 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
- we will try to use proprietary systems and software, or I can leave the comparison as an exercise to the students once back home

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, your users, or some other reason you do not (or do) control, 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 may be a simple thing, and I hope and wish you that after that much time using labs you are used to the computers here, the system, and their configuration. But if you ever catch me trying to read Cyrillic or even Hungarian, you will understand what I mean here: the things you are not used to and cannot do in autopilot mode, impede you and slow you down.

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. I have been collecting some other lecturers materials and noticed you had to do some Unix manipulation so far, so hopefully all this will not be so cryptic even though you may have some other paradigms at home. 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`, ... I will be preparing these things for you in the programs we will be using, but of course it is better for you to understand what is going on in order to be able to redo it on from a different scratch situation back home.

- 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 personnality.

- 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 should 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 a website listed from the conference documents or

my personal 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. I try to remember the time (not too long ago) when I was a beginner and I could not make sense of what my teachers were telling me. I was explaining myself things in another way and can try to customize the explanation to the person asking and her personal mental universe if I can enter it. 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.

We sadly still need to understand most of what is going on because in most cases XML technologies are still experimental and being deployed, and it's not always as transparent as one may wish it to be, even though it is getting better with time. As examples, the DocBook source code published was not even syntactically correct last year! The Linux Documentation Project and other documentation working groups are still busy migrating to XML technologies.

4 Examples of Unix shell manipulation

I will now be giving you a crash course about basic Unix manipulation. It may prove absolutely useless to you, absolutely impossible to understand, depending on your background. All of the following things may not be necessary to know and understand fully because I have to recognize that with time, the software is becoming more and more user-friendly (this meaning one is lured into thinking he does not need to learn anything in order to use a computer). But as always, it cannot hurt to understand what is going on, what can do the most, can do the least, and being Unix-literate helps understand what is going on when somethings goes wrong and helps find quick and cunning solutions to unexpected problems.

```
$ ls -al
```

This example shows how to use a common command, use shortcuts for its options (the single letter version of the options), and combine them several at a time. You can use either one or the long-option equivalents of these options.

```
$ apropos xml
```

This example shows how to look up for information related to one given keyword, hoping that the write of the manpage will have used that specific keyword in the short description of the command of course. Starting from there, one should of course be able to read and interpret the results and their jargon. One still hits the same old bootstrap problem.

```
$ man nsgmls | grep -2i variable
```

This example shows how to put different commands in queue one after another and combine them in apparently one single run.

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

This example introduces new things: the linebreak for continuation of the same command line, and the backquotes as the re-use of the output of some other command.

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

This example introduces the use of a loop in shell script language. This is for example the type of basic construct I use when I download digital pictures from my camera several times in the same day and need to change the numbers so that they follow up the numbers of the last batch.

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

This example shows how to locally set a variable so that it affects only the given command.

5 Example of a Makefile

```
ROOT_DIR=$(shell pwd)  
FILE=CEEnet2001.xml  
XSLT=$(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  
[...]
```

6 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. Think Unicode. Think QWERTY keyboard (for English-language typists at least). The inertia is tremendous.

- 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. Cyrillic, if you need to use it from time to time, can be fairly easy to remember as well (and why not decide on a Cyrillic-lock key then?).

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 some problem or some limitation. I have been recently informed that the newest desktop environments interfered with the graphic terminals in some nasty ways. 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 it is packaged and had 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 (the upcoming PDAs and other small devices are not bound to make the advances in memory technologies make that sort of overhead ignorable) and most of the programs are not really designed to support this in input.

7 What is XML?

The eXtensible Markup Language¹ 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, beta, or incompatible with the examples shipped with another piece of software. The activity on XML-related software makes one wonder about the technology: so many things come out so frequently it is difficult to keep up.

It is a meta-language.

¹<http://w3c.org/XML/>

"Meta-" is used here in as a computer science people jargon. A tool like Usenet was originally designed to make it possible for people to talk about their hobbies and interests. When Usenet newsgroups start to carry over endless threads dealing with Usenet policy or technology, then the tool just talks about itself and one may wonder if we are not all going crazy somehow.

The syntax looks like HTML. There are special keywords embedded inside angle brackets, with possible options, and some special characters are written in a special way, but the list of available options is not fixed and can be redefined for every document. The tag set is not fixed. It's just like agreeing on an alphabet but not on the words of the 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 written document, above all through the use of inline elements. All the high-level work and classification remains to be done by the author.

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? You still probably experience bugs in your usage of computers. 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.

8 The components of XML

We will not be using all of the subtleties of XML and all of the available components and options. As with most things, it suffices to know and understand a small subset of things to already be able to use a technology usefully (I still cannot use my new digital camera to its full power), and the amount of time we have at our disposal is limited.

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

These different components will be examined one at a time.

9 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"): <
- right angle bracket ("greater than"): >
- ampersand: &
- single quote (or "apostrophe"): '
- double quote: "

This kind of problem appears in all languages and formats, and is a kind of a reflexive, meta-level: how to mention a keyword of a language in that same language? One needs some kind of escaping or quoting character. How to mention that same quoting mechanism in the system? ...

10 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

11 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>Concept of XML</title>
  <conference>7th CEE-net workshop on networks technology</conference>
  <place>Budapest, Hungary</place>
  ...

```

12 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)

13 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>
```

14 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
- and (probably!) more... I am not sure I want to know

15 The goals pursued in the making of XML

(courtesy of "A Technical Introduction to XML"², by Norman Walsh)

²<http://www.xml.com/pub/98/10/guide1.html>

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

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.

- compatible with SGML for backward-compatibility reasons

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.

- 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.

- 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)³. 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.

16 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 (against some super-DTD!) with usual XML tools.

DTDs have no typing of data 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.

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

³<http://www.xml.com/pub/98/10/guide5.html>

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.

17 Syntax of a DTD

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

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

It is perfectly possible to not use every element defined in a DTD, and in particular to refer to a DTD for an element of just an inline or any other simple element. This might induce programs like Jade in default HTML mode, instead of writing the results of the transformation in one or several files, to spit it out on standard output so be careful (a `Makefile` of mine broke on such a matter).

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.

For written documents and materials, 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.

18 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>

```

Other DTDs exist, with many more elements and attributes and complicated syntax. One must not make the confusion between an XML document (which is necessarily a tree) and a DTD (where an element can refer to itself, or include several similar elements).

19 When to stop tagging?

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

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.

20 Encoding issues

The standard, default encoding for XML and XML tools is UTF-8⁴, 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

⁴<http://czyborra.com/utf/>

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):

```
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))/gei;
```

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
$format=$ENV{"UCFORMAT"} || '<U%04X>';
s/([\xC0-\xDF])([\x80-\xBF])/sprintf($format,
unpack("c", $1)<<6&0x07C0|unpack("c", $2)&0x003F)/ge;
s/([\xE0-\xEF])([\x80-\xBF])([\x80-\xBF])/sprintf($format,
unpack("c", $1)<<12&0xF000|unpack("c", $2)<<6&0x0FC0|unpack("c", $3)&0x003F)/ge;
s/([\xF0-\xF7])([\x80-\xBF])([\x80-\xBF])([\x80-\xBF])/sprintf($format,
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.

21 Lab on Concept of XML

In the laboratory session following up this presentation, you will do the following things:

- Learn how to check the well-formedness and/or validity of an XML document, and interpret the mistakes
- Edit and create by hand simple XML documents
- Edit and create with more advanced tools simple XML documents
- Come up with a DTD for a specific and simple application
- Create from data given in another format, valid XML documents for that DTD