Emaxml

An Emacs mode for editing XML

Paolo Debetto

Supervisor: Dr. Joe Wells

CS4 Dissertation

Deliverable one

Contents

1 Introduction **2**

1.1 Emacs 2

1.1.1 Editing modes 2

1.1.2 Emacs concepts 3

1.2 XML 4

1.3 Existing Emacs modes for editing XML 5

1.4 Similar systems 6

1.4.1 Conglomerate editor 6

1.4.2 GETOX 7

1.4.3 XED 7

2 Emaxml specification **8**

2.1 The look of Emaxml 8

2.1.1 Logical display units 10

2.2 The behavior of Emaxml from the user's point of view 12

2.2.1 Standard Emacs editing operations *à la* Emaxml 13

2.2.2 Emaxml specialized editing operations 18

2.3 Control issues 20

3 Development plan **21**

3.1 Main components 21

4 Performance assessment **23**

4.1 XML-equivalence of two documents 23

4.2 Testing the Parser and the Writer 23

4.3 Testing Emaxml interactively 24

4.4 Documentation requirements 24

5 Timetable **25**

6 Initial achievements **26**

Bibliography **28**

A Standard Emacs commands recoded for Emaxml **29**

B Emaxml commands **31**

C Notes on yanking transparency **32**

D Chronicle of a murder **34**

Chapter 1

Introduction

Emaxml is an extension of Emacs, written in Emacs Lisp, to edit XML documents. Major Emacs modes for editing SGML and XML already exist; this is different in that it allows viewing the document as a tree structure, both visually and logically.

1.1 Emacs

1.1.1 Editing modes

Editing any particular type of document requires very often a specialized editor that allows the user to perform some peculiar processing, or that automatically changes the document layout, or that displays the text in a way which is different from how the text is actually stored.

For instance, editing C code and editing a text document are two very different activities. Paragraph structure is not important when editing code; indenting each line according to its syntax is not important when writing a letter.

Emacs is not simply an editor, it is a code editor, a text editor, a LaTeX editor, a structured outline editor, a directory editor, a tar file editor, an email editor, and a hundred others, not least an SGML (and XML) editor. Emacs deals with each type of document by being in the appropriate editing **mode**.

The basic Emacs core consists of a set of capabilities such as managing buffers, windows, files, the cursor, etc., plus a Lisp interpreter, and was written in C. Emacs modes are extensions to the core, and are written in Lisp, or, rather, in Emacs Lisp, the Emacs dialect of Lisp.

1.1.2 Emacs concepts

I assume the reader of this document to be averagely familiar with the functioning of Emacs from a user's point of view. A few key Emacs characteristic features that are particularly relevant to my project are briefly defined here. For detailed information refer to the Emacs Info manual by pressing 'F1 i' in Emacs; the following definitions are mainly summarized from there.

- **Buffer**: the basic editing unit; one buffer corresponds to one text being edited. There may be several buffers, but at any time only one is being edited, the 'selected' buffer,
- **Frame**: an X Window System window in which Emacs is running. (The following definition for an *Emacs window* refers to subdivisions of one frame.)
- **Window**: Emacs can split a frame into two or many windows. Multiple windows can display parts of different buffers, or different parts of one buffer.
- **Point**: the location at which editing commands will take effect. In the current buffer, the cursor shows where point is.

  If several files are being edited in Emacs, each in its own buffer, each buffer has its own point location.

  A buffer that is not currently displayed remembers where point is in case it is displayed again later. If the same buffer appears in more than one window, each window has its own position for point in that buffer.

  The point is also one end of the *region* (see below).
- **Cursor**: The cursor is the rectangle on the selected buffer that indicates the position of the point. The cursor is on the character that follows point. Often people speak of 'the cursor' when, strictly speaking, they mean 'point'.
- **Mark**: an abstract pointer to a position in the text. The user can set it to specify one end of the region (see below), point being the other end. Each buffer has its own mark.
- **Marker**: a specialized Emacs internal data structure that defines a location in a buffer in terms of a pair $(buffer, location)$. It is worth noting that a marker follows the text as editing changes are made. Specifically, if text is deleted or inserted before the marker, the marker's position (an offset from the beginning of the buffer) is adjusted.

- **Mark Ring**: used to hold several recent previous locations of the mark, just in case the user wants to move back to them. Each buffer has its own mark ring; in addition, there is a single global mark ring.

- **Region**: The region is the text between point and the mark. Many commands operate on the text of the region. If a portion of text is highlighted with the mouse, that becomes the region and point and the mark are updated accordingly.

All these concepts, plus many others, are common to all Emacs applications and an Emacs user will expect Emacs to behave consistently in a new mode. Thus, the features of Emaxml must be designed to meet what a typical Emacs user would instinctively try to do in order to accomplish a task.

1.2 XML

I assume the reader to have a basic knowledge of XML. The purpose of this section is not to give an exhaustive description of XML, but to point out a few XML features that are important in the discussion of the details of Emaxml. For further information refer to [1], [2].

A good web page for quick basic information is at *http://www.w3.org/XML/1999/XML-in-10-points*.

- XML is a *syntax* which uses tags to allow tree structures to be written as a sequence of characters.

- An XML document is basically a *tree* structure, composed of a *root element* and (possibly) its *children*.

- The information in an XML document is meant to be processed by a *client application*, which may be very specific to a particular task. For example, a library managing application may keep the data base for books, users, etc. in XML format.

- The set of XML tags is not finite. A set of tags and their dependencies can be defined for each class of documents.

- Text consists of intermingled character data and markup.

  Markup takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white

space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).

All text that is not markup constitutes the **character data** (**CharData**) of the document.

- Some characters such as the '<' sign are not allowed in an XML document other than for markup and in comments and CDATA sections. Where these characters are needed, special sequences called **character references** are used. For example, '<' is the character reference for '<'.

As a matter of fact, XML was born and is mainly used to store structured information, even though its flexibility makes it also a highly powerful format for professional text editing.

1.3 Existing Emacs modes for editing XML

In Emacs 20.7.1, XML documents are edited under *SGML mode*, SGML being a predecessor of XML.

```
<?xml version='1.0' encoding="UTF-8"?>
<!DOCTYPE image-library SYSTEM "imagelib.dtd"
[
  <!ENTITY % imglib_ex SYSTEM "imglib_ex.ent">
  %imglilb_ex;
  <!ENTITY imglib_pic SYSTEM "imglib.gif" NDATA gif>
  <!ENTITY imglib_ent SYSTEM "imglib.ent">
]>
<image-library>
  <image id="emc2" filename="emc2.gif">
    <!-- E=mc2 in gif format -->
    <properties format="gif" width="162" height="89">
      <meta name="interlaced" content="no"/>
      <meta name="colors" content="256"/>
    </properties>
    <?graph_applic display_thumbnail?>
    <alttext>The equation E = m c-squared</alttext>
    <alttext-html>
      <![CDATA[<p>The equation &apos;E=mc2&apos;</p>]]>
    </alttext-html>
    &imglib_ent;
  </image>
</image-library>
```

Figure 1.1: Screenshot of SGML mode

In Fig.1.1 an XML file is being edited. The approach used by SGML mode is that of syntax highlighting. Tag names, attribute names, attribute values and the actual information (i.e. the *CharData*) are in different colors.

The tree structure of the document is not taken in account by Emacs, and the indenting is left to the user. For instance, the TAB key pressed in the middle of a line does not perform automatic indentation as typical in other Emacs modes, and pressed at the beginning of a line indents it as the previous line.

Many facilities are provided for manipulating elements.

Another Emacs mode suitable for editing XML documents is PSGML. It has additional features such as support for indentation which corresponds to the logical structure of an XML document.

PSGML is not part of the standard distribution of Emacs 20.7.1, but must be obtained separately.

1.4 Similar systems

Emaxml (as it should be when developed further) falls in the software category of non-commercial XML editors.

In particular, other similar existing systems may or may not offer a graphical view of the tree, and may or may not check that the user is building a tree consistent with the DTD.

Creating Emaxml is in my opinion justified by the fact that it would give Emacs a visual XML mode, which means that Emaxml is not just another XML editor, but an integrated part of the most powerful editor around. The quantity and quality of documentation available about programming Emacs also mean that Emaxml could possibly be perfected by anyone feeling so.

I have examined three similar programs, all from the open source community.

1.4.1 Conglomerate editor

Conglomerate [1] is not a simple XML editor. Actually, XML is not even mentioned in the web pages, from which the following description is extracted:

> "Conglomerate is a complete system for working with documents. It lets the user create, revise, archive, search, convert and publish information in several media, using a single source document."

This project has ambitious goals:

[1] Conglomerate home page is at www.conglomerate.org .

"To [reach out to] a wide audience, from would-be Word users to techies.

Simplify simultaneous publishing of information in a range of output formats (print and online) from a single source.

Replace the WYSIWYG document processing paradigm with a separated structure/appearance approach, even for simple tasks."

From Conglomerate I have taken the idea for the expanded view in Emaxml, as can be seen from figure 1.2.

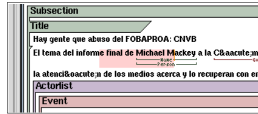

Figure 1.2: Conglomerate frontend

1.4.2 GETOX

"This software aims at giving users the ability to write XML files without having advanced knowledge of XML concepts. It should also allow users to produce valid documents at any time."

At the present stage of development, GETOX[2] allows the user to add and remove tags according to the DTD, edit text in PCDATA, and other editing operations. It does not support yet cutting/pasting parts of the XML tree and editing attributes.

1.4.3 XED

"XED[3] is a text editor for XML document instances. It is designed to support hand-authoring of small-to-medium size XML documents, and is optimised for keyboard input. It works very hard to ensure that you cannot produce a non-well-formed document. Although it does not validate, the results of offline validation can be accessed, and it does read DTDs and keep track of your document structure, and provides context-based accelerators to make element and attribute entry fast and easy."

XED offers facilities for editing the raw XML code.

[2] GETOX home page is at http://idx-getox.idealx.org/index.html

[3] XED home page is at http://www.ltg.ed.ac.uk/ ht/xed

Chapter 2

Emaxml specification

The main goal of Emaxml is to provide the Emacs user with a view of an XML document as a tree, and with a set of facilities for manipulating it handily.

This can be achieved by creating the Emaxml mode, which should hide the XML syntax by displaying the document in a pseudo-graphical, customizable, hierarchical fashion and automate the most common or most tedious actions involved in the editing of an XML document.

A very high number of useful editing functionalities can be devised for Emaxml mode. Moreover, an XML editor can enforce a certain degree of control over the semantics of the tree, i.e. can be aware or not of the DTD for the document being edited and help the user in a number of ways by exploiting this knowledge.

In sections 2.2 and 2.3 I arbitrarily set the initial target for my project in terms of functionalities and in terms of the level of control implemented, respectively. However, it is a strong requirement that the core of the mode be coded and documented in a way to facilitate later extension in both directions.

The definitions given in **bold** in this chapter do not refer to standard terminology, but are specific to Emaxml.

2.1 The look of Emaxml

Figure 2.1 shows what an XML document should look like when it is being edited with Emaxml[1]. It also shows the terminology for the logical units, which are the subject of section 2.1.1.

[1] For comparison, the portion of document displayed is the same as Fig.1.1.

Figure 2.1: Screenshot of Emaxml, with the indication of the Logical Units

The visual effect of a tree structure is obtained by using one color for each level of depth. Children are enclosed inside their parent, and attribute names and values are put together with the element name in the header lines for that element.

The topmost element is called the **seed element**[2], and does not correspond to an actual XML element; it contains information relative to the document, such as that contained in the XML declaration. From the user's point of view, the header of the seed element is treated as a normal header whose element name is the name of the file, and whose attributes are the said information. None of the fields contained in this header are compulsory [3], although they are very common. So they will appear by default when visiting a new document, but, if left blank, they will not be written in the file.

2.1.1 Logical display units

To describe some of the features of Emaxml, the concept of an Emaxml logical unit is introduced here.

A **logical unit** is a portion of an Emaxml buffer that corresponds to some XML syntactic component. At any moment during editing point will be in one or more logical units. Also, a particular location in the text, such as that defined by the mark, can be said to be in a particular logical unit.

The response of Emaxml to a user's input will vary depending on what type of logical unit the cursor is in at any moment, and the editing operations will have different meaning depending on whether they are performed on a region which is completely included in an instance of a logical unit or not.

The following are the possible logical units [4]:

- Element: a whole element, including its header and its children;
- Header: an element's name and its attributes;
- Element name: the first word of a Header;
- Attribute list: the set of attributes of an element and their values;

[2]Because it comes before the root element...

[3]In fact, the BNF production that defines the prolog is

$$[22]\ prolog ::= XMLDecl?\ Misc^*\ (doctypedecl\ Misc^*)?$$

[4]Throughout this document, logical unit names are in Sans Serif font

- Attribute: an attribute's name and its value;
- Attribute name: an attribute's name;
- Attribute value: an attribute's value;
- CharData: the plain text;
- Processing Instruction: a processing instruction;
- Processing Instruction Target: the target of a processing instruction, i.e. the first word of a Processing Instruction;
- Processing Instruction Body: the command of a processing instruction;
- Comment: a comment text;
- Internal DTD: the text relative to the definition of the internal DTD;
- CDATA section: the text of a CDATA section.

There may only be one instance of a Internal DTD and it must be a child of the seed element. In the text of the Internal DTD the user can actually write whatever s/he wants, since it is not parsed.

Element, Header, Attribute list Attribute and Processing Instruction are **compound** logical units, that is, they include smaller logical units(e.g. an Attribute list is composed of one or more Attributes, which in turn are composed of Attribute names and Attribute values). The remaining logical units are self contained and are called **elementary** logical units.

Not all locations in the display belong to a logical unit. For instance, the colored spaces at the left of an Element name or the colon following an Attribute name are not part of any logical unit. All such locations and their contents are said to be **automatic**, because they are managed by Emaxml and cannot be reached by the user. Non-automatic locations of the buffer form the **user space** of some logical unit.

Internal DTD, Comment, Processing Instruction and CDATA section are **special** logical units: their headers are automatic.

Special logical units and CharData are **multiline** logical units: they may contain *newline* characters and are displayed accordingly. The other elementary logical units are called **monoline**.

A **logical line** is either one line (up to a newline) of a multiline logical unit or an entire monoline logical unit.

The **previous** logical line with respect to a logical line is the first logical line that is encountered by going left and up. The **following** logical line with respect to a logical line is the logical line that is encountered by going right and down.

Logical units that correspond to those XML components which may be leaves of the tree (i.e. Element, CharData, Processing Instruction, Comment, CDATA section) are called **leaf logical units**.

Entity references will appear in Emaxml as part of normal CDATA, since they are considered only as syntactic constructs.

A portion of an Emaxml buffer delimited by two locations l_0 and l_1 both in user space is called a **unit**[5] and can be:

- a **string**, if l_0 and l_1 are both inside the same elementary logical unit;
- a logical unit, as said, if l_0 and l_1 are the first and the last location respectively of the same logical unit;
- an **illogical unit** [6], if l_0 and l_1 belong to two different logical units.

2.2 The behavior of Emaxml from the user's point of view

This section specifies Emaxml requirements. It is subdivided into two main parts: the description of what the common editing operations mean under Emaxml mode and the definition of the new editing operations specific to XML manipulation.

Editing levels

In editing an XML document, a basic operation (such as insertion or killing) can be performed at three different levels:

- At the **character level** the operation involves an Emaxml string.
- At the **logical level** the operation is performed over an entire logical unit, for example the deletion of an element.
- At the **illogical level** the operation is performed on an illogical unit.

[5]The term *unit* is used instead of *region* for consistency with the term *logical unit.*

[6]The adjective *illogical* is here chosen on purpose, to emphasize the apparent weirdness of a type of structure that instead may sometimes come quite handy.

2.2.1 Standard Emacs editing operations *à la* Emaxml

The typical Emacs user will expect a number of standard editing facilities to be available in Emaxml mode, and their behavior to parallel that of other modes.

As said, my project has as a target the redefinition of a finite set of operations only. This is a subset of the list obtained by pressing 'F1 b' in a *Fundamental* buffer in Emacs. It is not as important to implement a large number of functionalities immediately as it is to design the code in a modular fashion and document it properly, and choose at least a few operations from each of the following categories:

- **PM**: point movement;
- **ID**: insertion and deletion;
- **MK**: mark operations;
- **RE**: region setting;
- **SR**: search & replacement;
- **GE**: general common operations.

Appendix A lists the minimum set of commands to be implemented for Emaxml as part of this project.

Visiting a file

Emaxml mode will be activated on visiting a file with extension *.xml* and when a buffer containing a file with extension other than *.xml* is saved with extension *.xml*[7]. In the latter case, the buffer needs be re-displayed also. In case the file does not already exist, the buffer will contain the seed element, blank, as described in section 2.1.1.

Highlighting the region

When using the mouse to highlight the region, or in Transient Mark mode [8], only the locations of user space included in the region will be highlighted.

In figure D.1-(0) the region highlighted is *illogical*, i.e. it starts in an elementary logical unit and terminates in a different one.

[7] In Emacs, if a buffer is saved with 'C-x C-w' ('Save Buffer As...' in the Files menu) with a different extension, its major mode changes accordingly.

[8] In Transient Mark mode, when the mark is active, the region is highlighted.

Whenever a leaf logical unit is completely included in the highlighted region, all of its physical space will be colored, including the non-user space, otherwise only the included user space will be colored. In figure D.1-(0), the comment and the first meta element are examples of the first case, the image, properties and the last meta elements are examples of the second case.

Two useful commands will be implemented in Emaxml: mark-more and mark-less. The first is to expand the region to the next bigger logical unit containing the region (or containing point if mark is undefined). The second will do the opposite. For example, if the region is currently a string in an Attribute value, mark-more sets region to the containing Attribute. Further calls to mark-more set region to the containing Attribute list, then to the Header, then to the Element, and so on up to the entire buffer. Calls to mark-less will always select the first child when there is more than one.

Text search and replacement

Text search and replacement in Emaxml will operate on the user space only. Both string and regular expression search and replacement will be implemented.

Moving point

Movement in Emaxml acquires different meanings (and names) depending on which level is performed at:

- At character level: **sliding**.
  - **Horizontal sliding**: moving by one character.

    Point can slide inside the current logical line one character backward or forward.

    Sliding point backward when at the beginning of a logical line moves it to the end of the previous logical line.

    Sliding point forward when at the end of a logical line moves it to the beginning of the previous logical line.
  - **Vertical sliding**: moving by logical lines.

    When performing a vertical sliding movement (e.g. 'next-line'), point is expected to behave as closely as possible to the usual behavior. A movement that starts at the lth location of a logical line, and ends at a different logical line, will end at the lth location of the arrival logical line if that has at least l characters, or

at its last location otherwise. However, l is remembered for successive vertical movements. This Emacs standard property I will call **climbing transparency**, because it is proved for an editor by demonstrating that performing a vertical movement in one direction followed immediately by a vertical movement in the opposite direction, point will always return to the initial location.

Point can slide one logical line up or down with climbing transparency.

- At the logical level: **traversing**.

  Traversing (the XML tree) refers to moving from one XML component to another, hence the logical units involved are: Header, CharData, Processing Instruction, Comment, Internal DTD, CDATA section.

  Traversing will be implemented with no climbing transparency; point is always moved to the first character of the arrival logical unit.

  - **Horizontal traversing**[9]: moving hierarchically.

    Point can traverse left from one of the said logical units to its XML parent.

    Traversing right can only be done from a Header to the first child of the current element, if any.

  - **Vertical traversing**: moving to peers.

    Point can traverse from one of the said logical units (except the seed element, which has no peers) to the next or previous instance of the same logical unit found in the buffer, regardless of them being sibling.

Mark and the mark ring

The mark ring will be implemented in Emaxml, consistently with its standard definition and functionality. Obviously, the internals relative to such implementation will be specific to Emaxml; markers have to be objects of a different data structure, since they must describe a location in terms of the tree.

Killing and yanking in Emacs

Killing and *yanking* in Emacs jargon mean *cutting* and *pasting* respectively.

[9]The choice of adjectives *horizontal* and *vertical* for traversing is due to the fact that in an Emaxml buffer the parent of a logical unit can always be thought of as being “on the left”, while a peer is always up or down.

In standard Emacs operation when a portion of the buffer is killed it is deleted from the buffer and remembered by Emacs for later use. One property (I shall call it **yanking transparency**) of an Emacs buffer is that if some text is killed and immediately yanked, the buffer does not change. Point may be in a different location, though.

Yanking is an insertion operation; the portion of buffer being yanked is inserted before point.

Killing in Emaxml

Many Emacs commands exist for killing. The ones considered here are: kill-line, kill-region, kill-word, kill-buffer. Their names are self-explanatory. They are all implemented in Emaxml, the only one that changes its meaning being kill-line, which kills a logical line instead of a normal line. One further killing command is implemented in Emaxml, kill-element.

Since the object of the editing in Emaxml is a tree, the killing and yanking operations must be redefined in terms of logical units. For instance, what happens if the user tries to yank an Attribute list into a CharData?

Let us consider a portion p of an Emaxml buffer being killed that starts from location l_0 in leaf logical unit u_0 and ends at location l_1 in leaf logical unit u_1.

Emaxml will store killed logical units and illogical units as *trees* in the kill ring. For p a logical unit, the stored tree will be the tree rooted at u_0 and will include all its children. For p an illogical unit, the stored tree will be rooted at the smallest logical unit s that include all the logical units totally or partially included in p, minus the children of s not in p and minus the portions of user space of s not in p. This implies that the stored tree may have some blank logical units in it.

Killing p has the following effects:

(1) If p is a string, point is left at l_0.

(2) If p is a leaf logical unit then point is left:

- if p has a peer q below it, at the first character of q's user space;
- otherwise if p has a peer q above it, at the first character of q's user space;
- otherwise at the first character of p's parent's user space.

(3) If p is a non-leaf logical unit, then it is one of the elementary logical units in an

Element; a blank logical unit of the same type as u_0 is inserted in place of p with point at its first location.

(4) If p is an illogical unit, then point is left as in policies (1), (2), (3) but substituting "the portion of u_0 in p" for p.

When killing an illogical unit, Emaxml rebuilds the tree as follows, in order[10]:

(i) all leaf logical units entirely included in p are pruned;

(ii) all non-leaf logical units entirely included in p are blanked;

(iii) all characters in p are eliminated.

Yanking in Emaxml

In Emaxml, yanking will leave point after the last location of the user space of the yanked unit, i.e. the cursor will be under the first character of the user space after the yanked unit.

Yanking a non-string unit involves somehow tree manipulation. The only limitation posed by Emaxml is that non-leaf logical units cannot be yanked other than in a HeaderṠince both logical and illogical units are complete subtrees, there is no difference in yanking them.

Let us consider a unit p stored by Emaxml in the kill ring.

If p is a leaf logical unit, then yanking transparency does not hold [11], so p must be explicitly yanked as a peer or as a child. If yanked as a peer it is inserted before the current leaf, if yanked as a child it is inserted as its last child.

Yanking p into some logical unit u_2 is as follows:

p is a	**u_2 is a**	**Yanking p into u_2**
string	elementary logical unit	p can be yanked anywhere in u_2
leaf logical unit	Header	p can be yanked either as child or as peer of the element which u_2 belongs to
leaf logical unit	leaf logical unit	p yanked as a peer of u_2
Attribute list or Attribute	Header	p inserted at end of u_2's Attribute list

Figure 2.2: Yanking criteria for p not illogical.

[10]For an example, see Appendix D

[11]See Appendix C for a discussion of this.

2.2.2 Emaxml specialized editing operations

Appendix B lists the minimum set of new commands specific to Emaxml to be implemented as part of this project. They are described in the following sections.

Creating a new instance of a logical unit

Only new instances of a leaf logical unit or of an Attribute can be created.

Creating a new instance inserts a **blank** logical unit, described as follows:

- a blank Attribute name is represented by an automatic colon;
- a blank Attribute value is an empty string;
- a blank Attribute is a blank Attribute name and a blank Attribute value;
- a blank Attribute list is a blank Attribute;
- a blank Element name is represented by an automatic space;
- a blank Header is a blank Element name and a blank Attribute list;
- a blank Element is a blank Header;
- a blank CharData is three lines with white background;
- a blank instance of the remaining leaf logical units, is the appropriate lateral heading and a line with white background.

An Attribute can only be created when point is on the Header of a non-blank Element. If point is on another Attribute, this and the other ones are shifted down.

A leaf logical unit can be created as a child of an element or as a sibling of a leaf logical unit. A new child is inserted as the last child of the element, while a sibling is inserted in place of the leaf component.

Adjusting the displaying of the tree structure

An element (and the relative subtree rooted at it) can be displayed **outline** or **inline** (that is, vertically or horizontally) and **expanded** or **collapsed** (that is, completely visible or displayed as the element name only).

Outline Collapsed

Outline Expanded (default)

Inline Collapsed

Inline Expanded

Figure 2.3: Display modes

These characteristics are independent so there are four ways of displaying a subtree (see Fig.5.1), called **display modes**: outline-expanded, outline-collapsed, inline-expanded, inline-collapsed.

The **display state** of a subtree is defined by the display mode of all the elements it is formed by.

The following statements define the manipulation of the visual tree:

- The tree structure is by default displayed entirely outline-expanded when the document is initially visited.
- Making an outline subtree inline makes all its children temporarily inline, and does not change its or its children's expansion mode.
- Expanding a collapsed subtree brings it back to the display status it was before being collapsed (i.e. all its elements return to their previous display mode).
- The root element and the seed element cannot be made inline or collapsed.
- The Header of a collapsed or inline element is not user space.

An optional further development may be that the entire document display status be saved along with the file (e.g. encoded somehow inside the document) and restored the next time the document is visited in Emaxml mode.

2.3 Control issues

Emaxml, at the stage of development I set as target for my project, will not enforce control over the tree structure created by the user or read from an XML file in terms of the DTD.

Emaxml will see the document "simply" as a syntactic structure that must comply with the grammar set out by the BNF rules in [1]. Note that the internal DTD is *not* parsed. The user will be able to manipulate the tree as s/he wants without being bothered with indentation. The "less-thans", quotes, and other syntactic sugar of XML will be hidden. This should hopefully let the user concentrate more on the contents, but, on the other end, the user will also be able to create documents that are not well-formed, or invalid. The type of control enforced by Emaxml is syntactic only, but it is of a very high degree: the goal of the project is that it should be impossible to produce a document which does not comply with the BNF rules in [1].

However, the actual code implementing the editing mode must be designed so that adding semantic awareness and control should be easy.

Examples of events that may trigger a contents check or other semantics-related operations are:

- the contents of a non-leaf elementary logical unit (i.e. Element name, Attribute name, Attribute value) is changed at the character level;
- point leaves an elementary logical unit.
- the tree is changed; this includes creation of instances of any logical unit, killing and yanking at the logical and illogical levels,
- the contents of the seed element are changed;
- the contents of the Internal DTD are changed.

These and other such events must be easily recognizable and exploitable from the programmer's point of view.

Chapter 3

Development plan

3.1 Main components

The project is to be implemented as an extension to Emacs, i.e. as an Emacs mode. Thus, it is to be coded in Emacs Lisp.

The object of the editing, an XML file, will be seen by Emaxml in two ways at the same time: as an Emacs buffer (the **Ebuffer**), used to display the visual representation of the tree, and as a list (the **Etree**), structured as to allow an abstract, logical view of the document and appropriate manipulation.

The system has been divided in four major components, to be developed in sequence:

1 **Parser**.

The Parser takes as input an Emacs buffer in Fundamental mode (i.e. not in SGML mode, hence with no meta-information about syntax highlighting) containing an XML document and extracts the relative Etree. The Parser checks the syntax of the document and gives an indication of the error in case it is not correct.

Some details of the implementation of the Parser are given in section 6.

2 **Writer**.

The Writer carries out the opposite. It takes an Etree and produces an Emacs buffer. It assumes the Etree to be correct.

3 **Emaxml mode**.

Emacs is extended with the new mode. This includes the code relative to the interactive management of the Ebuffer and the Etree, from the basic user functionalities such as hitting a key to the more elaborate tree manipulation commands.

Some problems to be solved are to implement Emaxml as an Emacs mode are:

- How colors are managed by Emacs and how to control the display and the cursor. How to maintain the display up to date with the abstract representation held in the Etree.
- What is the data model for the Ebuffer, i.e. its data structures and manipulation functions.
- How to represent a location on the Ebuffer (e.g. what point will be like? What data structure will a marker be?) and how to map it to its corresponding item in the Etree.

Chapter 4

Performance assessment

4.1 XML-equivalence of two documents

Let us consider a buffer b_0 containing an XML document. Suppose the Parser processes b_0 producing the Etree e then the Writer processes e obtaining buffer b_1. The two buffers b_0 and b_1 need not be identical at the byte level, but the two XML documents in them must be **equivalent** in XML terms.

4.2 Testing the Parser and the Writer

The following criteria can be used to test Emaxml performance.

(i) The Parser will be tested on a fully-featured XML document, say d_0, predicting what the resulting Etree e_0 should be. A good example of such a document is "bookcase.xml", in [2].

(ii) The Writer can be tested on e_0. The document d_1 in the resulting buffer should be XML-equivalent to d_0.

(iii) To prove that the Parser and the Writer effectively carry out inverse functions, d_1 is fed to the Parser again. The resulting Etree e_1 should be *identical* to e_0[1]. This process can be represented diagrammatically as

$$\boxed{d_0} \overset{\text{Parser}}{\longmapsto} (e_0) \overset{\text{Writer}}{\longmapsto} \boxed{d_1} \overset{\text{Parser}}{\longmapsto} (e_1)$$

[1] Actually, if the Parser is proved correct, test (iii) proves XML-equivalence between d_0 and d_1 too.

(iv) The Parser will also be tested on one or more *incorrect* XML documents. Expected results will be compared with the actual ones.

Note that the Parser is not required to detect an end-tag matching a start-tag with a different tag-name as an error, since that is not specified in the BNF rules.

All tests (i)-(iv) can be automated by setting up a testing framework.

4.3 Testing Emaxml interactively

Once proved the Parser and the Writer correct, the major mode can be tested interactively.

A sequence of steps in terms of keystrokes tests a particular feature. A set of such sequences tests the major mode. Such set must be devised to cover all the features of Emaxml.

A sequence of keystrokes will be tested against an Emaxml buffer in known state, and the resulting buffer checked visually first (this part can be automated very little...), and logically then, by examining one or more of the resulting Etree Ebuffer and the file written by saving the buffer (this could in principle be automated, but may prove expensively long to set up).

The features to tests are those described in the Emaxml specification, in particular the categories listed in section 2.2.1 and the response to the editing commands listed in Appendix A and Appendix B.

As soon as a working prototype of Emaxml is available, prior Emacs users will be asked to use it and assess how well its goals have been achieved. Their feedback will allow some problems identified to be fixed.

4.4 Documentation requirements

Following the Emacs philosophy and spirit, Emaxml is meant to be an extendible system.

Useful and usable documentation is a key requirement. If only part of the target is implemented, but it is well documented, it will still be a partial success.

In particular, two types of documents are required: code documentation and Emacs on line help.

Chapter 5

Timetable

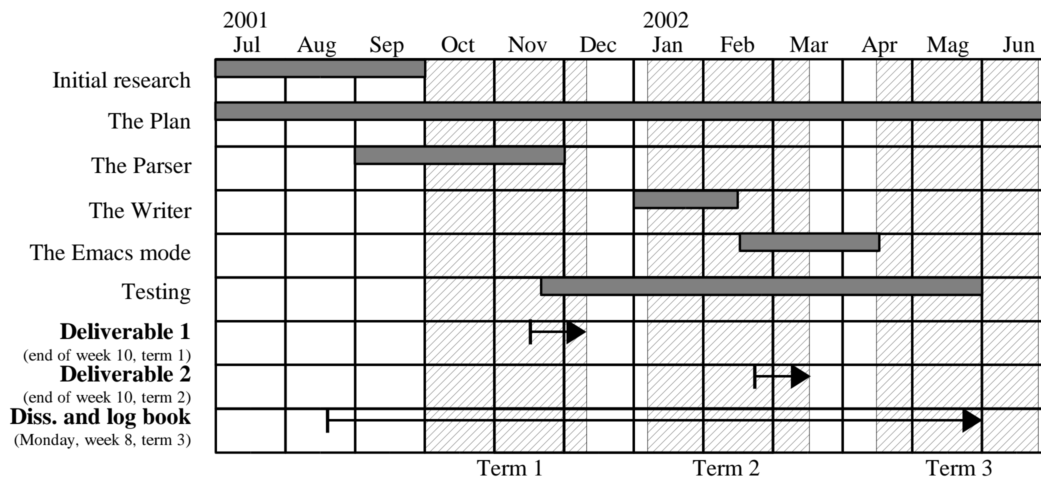

Figure 5.1: Tasks, deliverables and their dependencies

Chapter 6

Initial achievements

The work done so far have been along three major lines:

- Definition of the problem and initial study

  I have been reading on the three main subjects involved in the project, namely Lisp, Emacs internals and XML. The Bibliography covers the books and electronic documentation I have been through.

  Moreover, Emaxml has been defined in terms of the steps required to its development and the details of its features. Writing this document has clarified a great deal of such details.

- The Parser and the XMLDoc data model

  The Parser has been fully developed, but still needs testing. It is formed by a set of regular expressions matching the basic building blocks of XML, on top of which a set of functions is build that recognize and extract XML components.

  The output (Etree) is a list[1] that reflects the tree structure of the input document, in terms of objects of a data model that I have called XMLDoc (or XD for short). This data model has been derived from the BNF rules in [1] and has almost a direct parallel in the logical units model of the display.

- The documentation framework

  The documentation has been developed as a growing "Plan of work", a dynamic document that follows the development, describing the goals before they get implemented, and changing after the implementation to describe what has really been produced.

[1] As everything else in Lisp...

This somehow empirical approach to development has been taken due to my lack of experience with Lisp, Emacs and XML, and it has proven fairly successful.

All my work is kept in directory `~ceepd1/tesi` on the Department network. This is periodically updated with the work I do at home. The subdirectories contain README files that are updated with the help of a function in Emacs Lisp that detects the changes in terms of files deleted and added, and ask for the appropriate file description. The latest version of the Plan can be found at `~ceepd1/tesi/plan/Plan.html`.

Bibliography

[1] W3C XML Core Working Group. *Extensible Markup Language (XML) 1.0 (Second Edition).* http://www.w3.org/TR/2000/REC-xml-20001006, 2001.

This is the official specification of XML. Includes the BNF grammar describing the syntax of XML.

[2] E. Rusty Harold and E. Scott Means. *XML in a Nutshell.* O'Reilly, 2001.

Good discursive explanation of the basics of the various aspects of XML, plus a comprehensive coverage of all related topics and applications. I found it useful for initial documentation, and also as a quick reference.

[3] Free Software Foundation. *Emacs Info Manual.* Free Software Foundation, 1999.

Major source of information about the usage of Emacs. It is more than a help on-line; it can be searched in many ways and, as far as my experience is concerned, always answers one's questions. Moreover, it does not pop up unwanted saying that you are writing a letter.

[4] B. Lewis, D. LaLiberte and R. Stallman and the GNU Manual Group. *Emacs Lisp Manual.* http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html, 1993.

A book on Lisp, Emacs Lisp, Emacs internals, Emacs Lisp libraries. As readable as a novel, as useful as a quick reference. Available in a variety of formats including Info, which makes it embedded in Emacs.

[5] B. Glickstein. *Writing GNU Emacs Extensions.* O'Reilly, 1997.

Covers the customization of Emacs from the very basics of Lisp to a full major mode implementation. Very rich of practical examples paired with Lisp theory.

Appendix A

Standard Emacs commands recoded for Emaxml

<u>**Notes**</u>:

- Editing operations are here referred to by their Emacs Lisp names, due to lack of space. For an exact definition, where these names are not self-explanatory, use 'F1 f <command>' in Emacs.
- The **Cat** column refers to the editing categories listed in section 2.2.1.

Cat	Shortcut	Emacs Lisp name	Action
ID	C-d	delete-char	Delete the following character. No action at end of an elementary logical unit.
ID	ESC k	kill-sentence	Performs kill-element.
ID	C-y	yank	Described in 2.2.1. Perform “yanking as a child”.
ID	S-insert	yank	Described in 2.2.1. Perform “yanking as a sibling”.
ID	BACKSPACE	delete-backward-char	In the middle of an elementary logical unit, behave as usual. At beginning, behave as C-b.
ID	RET	newline	Multiline logical unit: add a newline at point.
ID	insert	overwrite-mode	Usual behavior.
GE	C-l	recenter	Usual behavior
GE	C-_, C-/	undo	Usual behavior
KY	C-k	kill-line	Monoline: Kill the rest of the current logical line.Multiline: also, if no non-blanks there, kill thru newline.
MK	C-@, C-SPC	set-mark-command	Usual behavior.
MK	C-x C-x	exchange-point-and-mark	Usual behavior.
PM	C-a	beginning-of-line	Move point to beginning of current logical line.
PM	C-b, ←	backward-char	Slide backward.
PM	C-↓	forward-paragraph	Traverse down.
PM	C-e	end-of-line	Move point to end of current logical line.
PM	C-f, →	forward-char	Slide forward.
PM	C-←	backward-word	Usual behavior, through user space.
PM	C-n, ↓	next-line	Slide to next logical line, with climbing transparency.
PM	C-p, ↑	previous-line	Slide to previous logical line, with climbing transparency.
PM	C-→	forward-word	Usual behavior, through user space.
PM	C-↑	backward-paragraph	Traverse up.
PM	end	end-of-buffer	Usual behavior.
PM	home	beginning-of-buffer	Usual behavior.
RE	C-x h	mark-whole-buffer	Usual behavior.
RE	ESC @	mark-word	Usual behavior.
RE	double-mouse-1	mouse-set-point	Usual behavior, but highlighting the region as described in section 2.2.1.
RE	drag-mouse-1	mouse-set-region	Usual behavior. Region as described in section 2.2.1.
RE	mouse-1	mouse-set-point	Usual behavior, but highlighting the region as described in section 2.2.1.
RE	mouse-2	mouse-yank-at-click	Usual behavior, but yanking done a la Emaxml (see section 2.2.1.
RE	triple-mouse-1	mouse-set-point	Usual behavior, but region set to the whole logical unit, and highlighting as described in section 2.2.1.

Figure A.1: New meanings for old commands

Appendix B

Emaxml commands

Lisp-like non-definitive name	**Action**
create-element-child	Create a blank Element as a child of the current smallest element. Described in section 2.2.2.
create-element-sibling	Create a blank Element as a sibling of the current smallest element. Described in section 2.2.2.
create-*logun*	Create a blank instance of a logical unit of type *logun*.
traverse-right	Traverse right.
traverse-left	Traverse left.
mark-more	Described in section 2.2.1
mark-less	Described in section 2.2.1

Figure B.1: New commands to be implemented

Appendix C

Notes on yanking transparency

Establishing yanking transparency for a leaf logical unit needs finding a suitable definition for:

(a) where point is left after killing a leaf logical unit p;

(b) how a leaf logical unit is inserted when point is at (a).

I have not been able to find a reasonable pair (a,b) for which yanking transparency holds.

Example

The following is an example.

A possible reasonable definition for (a) is:

(a1) If p has a peer q below it, at the first character of q's user space; otherwise if p has a peer q above it, at the first character of q's user space; otherwise at the first character of p's parent's user space.

In all case point will be in a Header after killing. Possible reasonable definitions for (b) are: When point is in the Header of an element r, a leaf logical unit p is inserted

(b1) as a peer of r above r;

(b2) as a peer of r below r;

(b3) as the first child of r;

(b4) as the last child of r.

(A)

(a1,b1) on <branch2>
(a1,b2) on <branch1>

(a1,b3) on <branch1>

(a1,b4) on <branch1>

Figure C.1: Applying the definitions

Consider subtrees in figure C.1.

Starting from tree (A), pairs of rules are applied to one of *root*'s children. The trees other than (A) show where the pairs of rules fail with respect to yanking transparency.

Appendix D

Chronicle of a murder

As described in section 2.2.1, the three steps to obtain the resulting tree after killing p are:

(i) all leaf logical units entirely included in p are pruned;

(ii) all non-leaf logical units entirely included in p are blanked;

(iii) all characters in p are eliminated.

Figure D.1 shows what a highlighted region looks like, as described in section 2.2.1, and what the effects are of each step in the algorithm. The bottom right picture is the final result.

Step (i) prunes the comment and the *meta* element, which are leaf logical units entirely included in p.

Step (ii) blanks the Header of the *properties* element, and the Element name and the Attribute name of the second *meta* element, which are non-leaf logical units entirely included in p.

Step (iii) eliminates "c2.gif" and "co", which are all the characters in p.

Figure D.2 is a visual representation of the tree stored by killing p.

(0) Highlighting the region

After step (i)

After step (ii)

After step (iii)

Figure D.1: Killing algorithm steps

Figure D.2: The stored tree