Emaxml

An Emacs mode for editing XML

Paolo Debetto

Supervisor: Dr. Joe Wells

Academic Year 2001/2002

CS4 Dissertation

Acknowledgements

This work is the final product of four years of study at Heriot-Watt University. It is dedicated to nonna Olga, who knows me better than I do.

I would like to thank the following people, because without their help this would not exist, since I would probably have given up long time ago:

My parents, for believing in me.
My sisters, for loving me.
My brother in law, for playing Risk with me.
The Italian Long-Term-Tourists Support and Survival Group, Edinburgh, composed of (in chronological order): il Conte Nardi, Giovannone, la Bambola, Disguido, Paciugo, la Carletta & the Nikos, the Polper. And the others, too. These people can cure homesickness.
My supervisor, Joe Wells, for the things I have learnt from him.
My second reader, Emanuele (Manuel) Trucco, for reading what it's probably the most boring dissertation he's ever seen and not penalizing my delays (well, so far, at least).
Amelia Viorela Rastei and Peter King, for being so kind to Emaxml.
All the people who devised and realized Emacs, for their vision.

Contents

Part I

Introduction

Chapter 1

Background

"Revolutionaries are more formalistic than conservatives."
Italo Calvino, *The Baron in the Trees*, ch.28

> 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.

An XML document is often generated automatically by an application. Nevertheless, in many occasions XML code is edited directly by a human author. When a normal text editor (i.e. one with no XML-specific editing facilities) is used to this end, the author's creativity has to deal with the XML document at three levels:

1. At the **contents level**, the author is concerned with what the document is about, the actual information or concepts.

2. At the **structure level**, the author organises the document hierarchically, according to the rules set by the DTD for that particular class of documents.

   For non-trivial documents the overhead activity involved with keeping the structure in order or with changing the current structure can be very expensive.

   Moreover, the author has to be concerned with indentation or some other means to visually see the structure of the document.

   However, this activity is related to the conceptual contents of the document.

3. At the **syntactic level**, the author is concerned with getting the XML syntactic sugar right. This activity is strictly XML-related and has nothing to do with the topic of

the document. It is an error-prone activity and the overhead involved can be very expensive.

Obviously most of this work can be automated to various degrees by an editor with XML editing facilities, to the purpose of letting the author concentrating on the contents and the structure of the document abstractly.

The approach of Emaxml is that of taking care of the XML syntax and providing means of seeing and manipulating the structure of the document effectively, by displaying the document in a tree-like fashion.

Figures 1.1 and 1.4 show the same document, as it is on disk, and in Emaxml mode. Note that, as explained later, in Emaxml the characters involved with the XML syntax are managed automatically; *the user writes only contents-related and structure-related text.*

The concrete objective of my project is to implement a fully functional Emacs mode, with a limited number of functionalities, but designed so that new functionalities and features can easily be added by anyone who might possibly want to work on it later. Emaxml is therefore to be considered the inititation of an open source project or, at least, an investigation on what the issues may be in such a project.

1.1 Overview 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[1].

XML is a *syntax* which uses tags to allow tree structures to be written as a sequence of characters. Thus, it is used to store data of any kind in a standardized format.

An example of some XML code[2] is in figure 1.1

It is a book, divided in chapters and sections, with figures and quotations. The XML file looks exactly like that, and it is meant to be processed by some typographic **client application** which will likely produce an output in some format such as LaTeX.

The text of the XML document consists of intermingled character data and markup. The **markup** is enclosed in pairs of angular brackets or between '&' and ';' and is mainly concerned with the *structure* of the document, the rest is **character data** and represents the contents of the document.

[1]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*

[2]Throughout this document, the line numbers in square brackets are not part of the code.

```
[01] <?xml version="1.0"?>
[02] <!DOCTYPE book SYSTEM "./book.dtd">
[03] <book title="Structure and Interpretation of Computer Programs"
[04]       author="Harold Abelson"
[05]       author="Gerald Jay Sussman"
[06]       isbn="0-262-01077-1">
[07]
[08]   <?typ-appln make-index?>
[09]
[10]   <!-- Insert acknowledgements here -->
[11]
[12]   <chapter title="Building Abstractions with Procedures">
[13]     <quotation author="John Locke"
[14]                source="An Essay Concerning Human Understanding">
[15]       The acts of the mind, wherein...
[16]     </quotation>
[17]     We are about to study the idea of <em>computational
[18]     process</em>.
[19]     ...
[20]     <section title="The Elements of Programming">
[21]       A powerful programming language...
[22]     </section>
[23]
[24]     <section title="Procedures and the Processes They Generate">
[25]       We have now considered the elements of programming:...
[26]       <figure source="/path/factorial.eps"
[27]               caption="A linear recursive process for computing 6!."/>
[28]     </section>
[29]       ...
[30]   </chapter>
[31]
[32]   <chapter title="Building Abstraction with Data">
[33]     <quotation author="Hermann Weyl"
[34]                source="The Mathematical Way of Thinking">
[35]       We now come to the decisive step of...
[36]     </quotation>
[37]     We concentrated in chapter 1 on computational processes...
[38]     <section title="Introduction to Data Abstraction">
[39]       When we discussed procedures in section...
[40]     </section>
[41]     ...
[42]   </chapter>
[43] </book>
```

Figure 1.1: The file `book.xml`

The main components of the structure are the **elements**. An element is enclosed in a pair of **tags** (start-tag and end-tag) such as and in lines [17] and [18]. The **element name** is the first word of the start tag. A start tag can also carry some further information about its element, in form of **attributes**; for example, lines [13] and [14] say that what follows them is a quotation by John Locke, taken from "An Essay Concerning Human Understanding". Whatever is between a start-tag and an end-tag is contained in the element; there may be chunks of character data, other elements, or other markup structures, so the result is a recursive tree structure. There may also be nothing inside an element, in

which case it is called an **empty element**, and takes the syntax of an empty element tag such as <`figure .../`> in lines [26] and [27].

Thus, XML is tag-based, like for example HTML. Two great differences between the two formats are that in XML the set of tag is unlimited, and a pair of tags does not carry only information about the layout of what is enclosed, but also about its *meaning* in the context.

The other main markup components are:

- **Comments**, which are meant for the human reader and contained in a tag of the form <`!-- .... --`>, such as in line [10].
- **Processing Instructions (PIs)**, which are instructions to be executed by the client application at a certain point in the processing of the XML document. PIs take the form <`?target body?`>, as the one in line [08], which tells the typographic application to make the index automatically at that point of the book.
- **Entity References**, which are a sort of macro expansion facility, and have the form `&...;`.
- **Character References**, of the form `&#...;`, which expand to characters which are not in the keyboard.
- The **XML Declaration**, which states some information about the file, as in line [01].
- The **Document Type Declaration** (line [02]), which states the name of the **root element** (*book*) and the location of the file containing the DTD (described below).

What relates the XML document to the application is the meta-description of the structure underlying a book. This information is contained in an auxiliary file, called the **DTD (Document Type Definition)**, which defines the structure of all XML documents of class *book*. It will contain information such as "A book has a title, zero or more authors and an ISBN code, and is composed of character data intermingled with elements called chapters. A chapter has a title and is composed of ...." and so on, written in a syntax which is specified with XML in [1]. The DTD file is addressed in the Document Type Declaration.

To conclude, figure 1.2 depicts the stages a book goes through to be completed, and the role played by Emaxml.

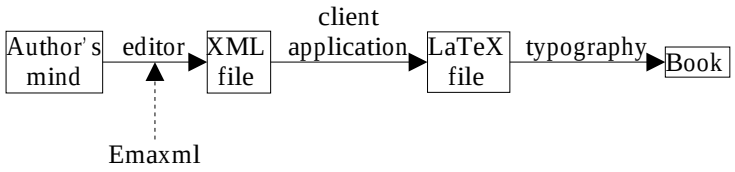

Figure 1.2: The stages a book stored in XML may go through.

1.2 Overview of Emacs

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.

1.2.1 Emacs Concepts

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**: an area of memory in which one text being edited is stored. What is displayed when a text is edited is the meaning of its buffer.

  A buffer is organized in **buffer cells**, each of which contains information such as what character is in that cell, its layout, its behavior under certain circumstances, and any information the current mode needs to attach to it. These are called the

text properties of that character.

There may be several buffers, but at any time only one is being edited, the 'selected' buffer. When a buffer is displayed, what the user can see is the representation of the bytes in the 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 of a buffer 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.

  One important property of the point is that it is *between* two characters.

  The point is also one end of the *region* (see below).

- **Cursor**: The cursor is the rectangle or the vertical bar 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 a buffer. 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.

- **Commands**: operations that the user can perform, as opposed to non-interactive Lisp functions. Commands include operations concerned with file, buffer, window and frame management, text processing of any kind, etc.

  There are many ways in which the Emacs user can run an editing command. The most common are keys, menus and the *minibuffer*.

- **The Minibuffer**: an area at the bottom of the frame, used to read in commands and command parameters.

  By hitting `'M-x'`[3] the user accesses the minibuffer and can type the name of a command followed by the Enter key.

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.

1.3 Existing Emacs Modes for Editing Xml

In Emacs 21, XML documents are edited under *SGML mode*, SGML being a predecessor of XML.

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.

[3]The notation used in Emacs for combinations of keys uses `'C'` for the Control key, `'M'` for the Alt key, and `'S'` for the Shift key. For example, `'C-a'` is Control and A, `'M-x'` is Alt and X, and `'C-M-w'` is Control, Alt and W, while `'C-x C-f'` means to hit first `'C-x'` and then `'C-f'`. The Enter key is indicated by `'RET'`.

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 21, but must be obtained separately.

1.4 Similar Systems

Emaxml (as it should be when developed further) falls in the software category of non-proprietary 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[4] 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:

> "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."

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

From Conglomerate I have taken the idea for the expanded view in Emaxml, as can be seen from figure 1.3.

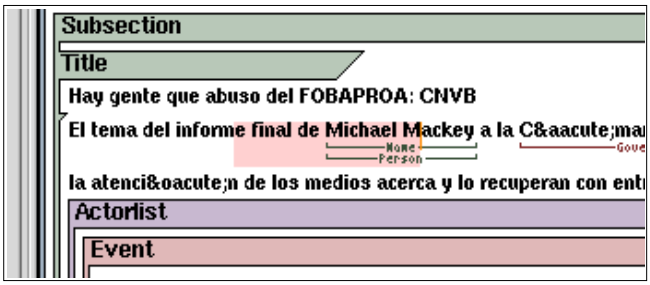

Figure 1.3: Conglomerate frontend

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[5] 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[6] 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.

[5] GETOX home page is at http://idx-getox.idealx.org/index.html
[6] XED home page is at http://www.ltg.ed.ac.uk/~ht/xed

Figure 1.4: The file `book.xml` being edited with Emacs in Emaxml mode

Chapter 2

Topography of Emaxml

This chapter provides a language for describing concepts related to an Emaxml buffer.

The display of an Emaxml buffer is based on a hierarchy of **logical units**, gathered when parsing the file at the activation of the mode. Each logical unit is an abstract area of the display that corresponds to a subset of the parse tree[1], so at any instant the cursor will be in a subset of all the existing logical units. Figure 2.1 depicts an example of this model.

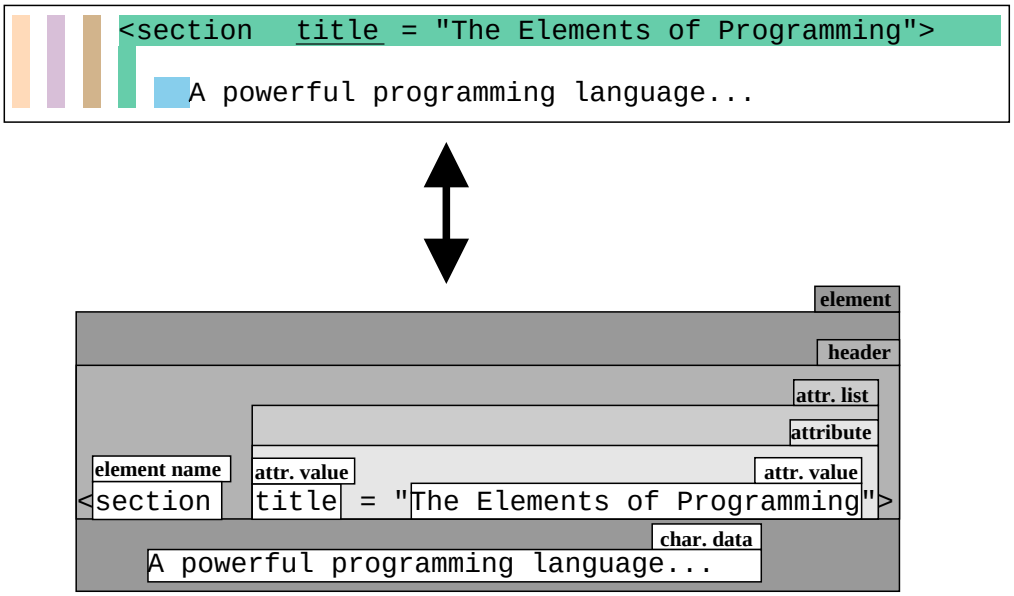

Figure 2.1: Example of hierarchy of logical units. It is an Element whose children are its Header and a Character Data. The Header's children are the Element Name and the Attribute List; the Attribute List has one Attribute as a child, which in turn contains an Attribute Name and an Attribute Value.

[1] See section 5.5 for further detail on parsing.

A logical unit is said to be **contained** in another if its boundaries are within that logical unit. The contained logical unit is a **child** of the containing one, which is the **parent**.

2.1 Types of Logical units

The **type** of a logical unit determines its layout and its behavior in response to user input, as well as the relationship it can have with other logical units.

The following is the list of logical unit types, which also shows the idea of hierarchy (but not all the possible parental relationships[2]).

Seed[3]

- Document Type Declaration
  - Root Name
  - External ID
    - Public ID
  - Internal DTD
- Element
  - Header
    - Element Name
    - Attribute List
      - Attribute
        - Attribute Name
        - Attribute Value
  - Processing Instruction
    - Processing Instruction Target
    - Processing Instruction Body
  - Comment
  - Character Data
  - Entity Reference

The choice of which types to include in the hierarchy has been mostly influenced by the BNF definitions of the grammar that specifies XML in [1].

Some characteristics of a few logical units are worth being specified:

[2] See table C.1 for a definition of the relationships between logical units.

- The most comprehensive logical unit is the Seed. It spans the whole buffer and contains all other logical units in the buffer.

- There may be only one Seed and one Document Type Declaration in a document. They always contains at least an instance of all the respective children even if they are empty, and cannot be deleted or added.

- The Seed must have one and only one instance of an Element as child, which is the **root element**. It cannot be deleted or added. The Seed, however, can have any instances of Processing Instruction or Comment as children.

2.2 Categories of Logical units

Logical units are further classified into categories, to group types with similar characteristics under a meaningful name.

2.2.1 Elementary vs. Compound Logical units

Elementary logical units cannot contain other logical units, in other words they are the leaves of the tree structure of the document.

Element Name, Attribute Name, Attribute Value, Character Data, Processing Instruction Target, Processing Instruction Body, Comment, Entity Reference, Root Name, Public ID, System ID and Internal DTD are the elementary logical units.

The main purpose of identifying this category is because most of the action in an Emaxml buffer happens in some elementary logical unit, since that is where the cursor is at any instant (see section 2.3). The elementary logical unit point is in at a particular moment is called the **current logical unit**.

Logical units that are not elementary are called **compound**. They are composed of other logical units, either elementary or not.

For example a Header is composed of an Element Name and an Attribute List. While the Element Name is elementary, the Attribute List is in turn made up of a series of Attributes, composed of their respective Attribute Name and Attribute Value.

When talking about logical units, "elementary" and "compound" can be omitted if that is unambiguously clear from the context. For example "...adding a character to an empty logical unit..." implies that the logical unit in question is elementary, since a compound logical unit has no characters of its own.

2.2.2 Primary Logical units

The main components of the structure of an XML documents are the Seed, the Document Type Declaration, and then Elements, Comments, Processing Instructions, Entity References and instances of Character Data. For this reason logical units of all this types are said to be **primary**.

An instance of a primary logical unit is called a **branch**. At any instant point is in all the branches containing it, up to the Seed, and the smallest (or most specific) of them is the **current branch**.

2.2.3 Multiline vs. Monoline Logical units

As said, the text editing in an Emaxml buffer happens only in elementary logical units. The contents of some of them are constrained by the syntax rules not to contain newline characters. For example, an Attribute Name can only be one word, with no whitespace at all.

These logical units form the category of **monoline** logical units, and are: Element Name, Attribute Name, Attribute Value, Processing Instruction Target, Processing Instruction Body, Entity Reference, Root Name, Public ID, System ID and Internal DTD[4].

On the other hand, some logical units are by their nature actual pieces of text, with whitespace and newlines in particular. These are called **multiline** logical units, and are all the remaining elementary logical units: Internal DTD, Comment, Character Data.

The editing process of a multiline logical unit is very different from that of a monoline one, and should be programmed to be as similar as possible to the normal editing of text in Emacs, with as many of the usual facilities as possible.

A **logical line** is one line of an elementary logical unit. Since monoline logical units have only one line, a monoline logical unit is also a logical line. The concept of logical line is useful to describe editing functionalities that apply in both monoline and multiline environments.

2.3 Buffer Space

Emaxml mode is based on the idea of a *controlled buffer*, in the sense that the cursor is constrained to move only over certain buffer cells (which form the **user space**), while the

[4]Actually, this is a bug in the present version of Emaxml, since Processing Instruction Bodies and Attribute Values *can* contain newlines.

rest of the buffer (the **automatic space**) is managed by Emaxml.

Another form of control Emaxml has over the buffer is that it spontaneously inserts some empty logical units in places where there should be one, even if the parsed file does not have them. For example, a Document Type Declaration is always present with all its children, and an empty attribute follows an Element Name if that element has no attributes (see the `'em'` element in figure 1.4). This feature adds data to the buffer that is somehow redundant, and should be made optional, in case the user does not like it. For this reason, such empty logical units are called **redundant children**.

Automatic space is composed of:

- **Sidebars**: the colored vertical bars on the left of the window, which give the idea of depth into the tree or tell which type of logical unit follows (e.g. the colored "<!" before a comment).

- **Syntactic sweeteners**: the characters that imitate the "syntactic sugar" of XML, although not completely. Example of syntactic sweeteners are the double quotes surrounding an Attribute Value, the "<" and ">" surrounding a Header, the words "DOCTYPE", "PUBLIC" and "SYSTEM" in the Document Type Declaration.

- **Semiautomatic characters**: read-only buffer cells on which the cursor can actually be. Their purpose is to allow the insertion of new characters at the end of a sequence of zero or more user characters.

  For example, the cursor can go on[5] the '"' after an Attribute Value. If a character is typed there, it becomes part of the Attribute Value. If the `'delete'` key is hit there, nothing happens.

  Note that there is a semiautomatic character at the end of *every* logical line, including those of a multiline logical unit[6].

  A semiautomatic character is said to **serve** the elementary logical unit it follows.

In practice, an Emaxml buffer can be thought of as a large inaccessible area with "holes" whose contents can be edited.

[5] If the cursor is displayed as a vertical bar, as by default in Emaxml, it will actually be *before* that character. This reflects better where *point* is, since point is always between two characters of the buffer.

[6] In fact, at the end of a multiline logical unit there is a semiautomatic space.

Part II

Emaxml from the User's Point of View

Chapter 3

Features and Functionalities

"Cool!"

Calvin, by B. Watterson

> This chapter offers a functional specification of the system as it is presently. Some ideas for possible improvements and suggestions on how to implement them are given in chapter 8.

It must be noted here that only part of the features of Emaxml that were identified in the design stage have been implemented; more ideas have come about since, and more will possibly come about later[1].

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 is 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.

The functions that redefine standard Emacs commands are bound to the keystrokes that perform the same commands when Emaxml is activated[2]. This ensures that the user will have to hit the same keys they are used to, even if they changed them from the standard definitions.

[1] I think that, being Emaxml meant to be an extensible system, the point is how *feasible* it is to add a new feature. I have concentrated on making the code be a set of tools and data structures for the programmer to reason at a high level rather than on providing a closed set of features immediately.

[2] Using `'substitute-key-definition'` with respect to the current global map. In the next version of Emacs, not yet released, there is a new "remap" mechanism which is better.

In this chapter the notion of *cursor* and that of *point* are used interchangeably. In fact, if the cursor is shaped as a bar it represents point visually.

3.1 Activating the Mode

Emaxml is started by visiting[3] an XML file (this will open it with the default mode for XML files, e.g. SGML mode), and then activating Emaxml mode with `'M-x emaxml-mode RET'`.

Figure 1.4 shows the `book.xml` file in Emaxml mode.

As described in section 2.3, the buffer will not reflect the file exactly, since some empty logical units are added automatically as redundant childrens.

3.2 Point Movement

Point is constrained to move around user space only. Some standard keystrokes for point movement have been bound to perform similar functions in Emaxml.

- Moving point to the next or previous character when at a logical unit boundary takes point to the next/previous character in user space, i.e. to the next/previous logical unit.
- Moving point to the beginning or the end of the line takes point to the first/last user character of the line, whether in the same logical unit or not.
- Movement by sentences has been substituted with movement by logical units, i.e. the keystrokes for the commands `'forward-sentence'` and `'backward-sentence'` move point to the beginning of the next logical unit and to the end of the previous logical unit respectively. Note that "next" and "previous" are here used in the physical sense, not in terms of hierarchy.
- Scroll commands work as usual.
- Moving point to the beginning or end of the buffer takes point to the first/last character of user space in the buffer.

[3] To *visit* a file, in Emacs slang, means to open it. It can be done from the File menu, or by hitting `'C-x C-f'`.

3.3 Elementary Editing

- Insertion of single characters works as usual. The character is added to the current logical unit. Insertion of a character on an empty logical unit and at the end of a logical unit occurs when the cursor is on the semiautomatic character serving that logical unit.

- Deletion of single characters works as usual backward and forward, but the following exceptions apply:

  – At the beginning of a logical unit, a backward deletion takes the cursor to the last character of the previous logical unit and deletes it.

  – At the end of a logical unit, a forward deletion does not do anything.

3.4 Killing and Yanking

Killing and yanking are Emacs terminology for "cutting" and "pasting". In standard Emacs they operate on regions of the buffer, which are stored as strings of characters. When a region is killed it is saved in the kill ring, which is a sort of circular list. Therefore, not only the last region can be retrieved, but also the other ones previously killed (up to the kill ring maximum size).

A region of the buffer is defined by two buffer position. This determines a string in a usual buffer, but in an Emaxml buffer the meaning of a region is more complex since it may involve the notion of subtree[4].

Killing, yanking and "copying as killed" have been implemented for regions that are contained in one elementary logical unit, whether monoline or multiline, because such regions can be simplified as strings. However, yanking a multiline string in a monoline logical unit has the effect of yanking up to and excluding the first newline character of the saved string.

The Emacs standard command `'kill-line'` is implemented for both monoline and multiline logical units, and parallels the usual behavior.

[4]For a discussion of this, see section 8.4.3

3.5 Adding Branches

A very common task is that of adding a new branch. With respect to the current branch, a new one can be added as a sibling (in which case it is inserted after the current branch, at the same depth in the tree) or as a child (that is, as the *first* child).

An instance of a new branch of whichever type takes the form an **empty branch** of that type, that is, the automatic and semiautomatic characters related to such a branch at the appropriate depth. This also include the possible empty instances of its redundant children.

The prefix key sequences `'C-c a'` (for "after this one") and `'C-c c'` (for "child") introduce the insertion of a new branch as a sibling and as a child of the current branch respectively.

They are followsd by a character that identify the type of logical unit ton be added, as follows:

& Entity Reference

! Comment

? Processing Instruction

c Character Data

e Element

For instance, to add a new Processing Instruction as a child of the current branch, the user hits `'C-c c ?'`.

A new Element, sibling of the current element, can also be added typing `'RET'` when the cursor is on the Element Name.

Adding new branches is the base of adding new material to the document. The way it is implemented now is somehow unnatural and unpractical. A couple of ideas to improve this are discussed in section 8.3.5.

3.6 Deleting Branches

The current branch can be deleted by typing `'C-c d b'`; it will not be saved.

3.7 Cutting, Copying and Pasting Branches

The operations of cutting, copying and pasting entire branches are separated from killing and yanking operations, because they operate on subtrees[5] instead of strings. A further difference is that there does not exist a ring to store cut or copied branches: only the last one is remembered.

The keys to perform these operations on the current branch parallel the killing, yanking and copying standard bindings, prefixed with `'C-c'` and, possibly, suffixed with an `'a'` or `'c'` for "after this" and "child":

- `'C-c M-w'` to copy
- `'C-c C-w'` to cut
- `'C-c C-y a'` to paste as a sibling of the current branch
- `'C-c C-y c'` to paste as a child of the current branch

3.8 Adding and Deleting Attributes

Addition of a new empty attribute is performed by hitting `'RET'` on an attribute. The new one appears below the current one.

Deletion of an attribute is performed by hitting `'C-c d a'` on the unwanted attribute. Note that an attribute cannot be deleted if it is the only attribute of the current Element, because it is a (possibly non-empty) redundant child.

3.9 Error Management

Emaxml performs low-level syntactic control over the contents of the current logical unit any time a change occurs.

Low-level syntactic control concerns only the contents of a logical unit independently of the DTD. What is checked is the well-formedness of the logical unit; for example, a Comment cannot contain the string `'--'`, an Attribute Namecannot contain a space, and so on.

In practice, the logical unit is parsed when its contents change, and if an error is found the logical unit is highlighted using a yellow background and a message is issued in the echo

[5] Although only *complete* subtrees, as opposed to the *incomplete* subtrees discussed in section 8.4.3.

area. The error message is memorized and can be retrieved by placing the cursor on an highlighted logical unit and hitting `C-c e`. The logical unit is displayed as normal again and the message forgotten if a change makes the contents legal again.

Emaxml also performs **low-level structural control** over some compound logical units. For example, an empty Element Name makes the whole Element erroneous unless it is empty.

3.10 Saving

The file can be saved in the usual way.

Note that if the buffer contains errors it is saved anyway, probably causing a parse error the next time it is opened in Emaxml mode. This is a bug that must be fixed, as suggested in section 8.3.7.

Part III

Emaxml from the Programmer's Point of View

Chapter 4

Introduction

"You must remember this
A kill is just a kill
A yank is just a yank
The fundamental mode applies
As lines go by"

> 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 is an XML document. As it is on disk it is a sequence of bytes, but it represents a structure that can be modeled as a tree: a *prolog*, a *root element* and an *epilogue*[1]. Using the logical unit model, the entire structure becomes a Seed whose children are, in order, the branches in the prolog, the root element, and the branches in the epilogue.

A data structure is needed to represent internally a tree of objects corresponding to the logical unit model; this is provided by the Etree. The Etree corresponding to an XML document needs to be parsed out of an XML file, manipulated and finally written into an XML file. These are the facilities offered by the XD Data Model, the first major software component of my project.

The Emaxml mode, on the other hand, provides a display representation of the Etree by reproducing it on an Emacs buffer structured in a particular way (an Ebuffer), and by performing on it the editing operations requested by the user.

In conclusion:

[1]The term **epilogue** is not part of the specification of XML. It refers to the branches following the root element, that can be of type Processing Instruction or Comment.

- The XML file, the Etree and the Ebuffer are three different concrete ways of representing the same abstract object, the XML document.
- The XD Data Model relates the file to the Etree, and allows manipulation of the latter.
- The Emaxml mode relates the Etree to the Ebuffer, and allows manipulation of the latter by the user.

The following priorities have been kept in mind during coding:

- Consistency with the XML specification given in [1], in particular with the BNF definitions numbered in square brackets (**BNF-defs** for short).
- Modularity, so that functions and constants can be re-used in different contexts and extended easily.
- Readability of the code, to facilitate future possible improvement.

Chapter 5

The XD data model

The idea behind XD is to provide an independent, reusable tool for parsing, writing and manipulating XML files according to the XD data model.

In some respect, the XD data model is quite limited, since it does not cover all the aspects of an XML document to a high degree of detail, but it can be useful for any application that, like Emaxml, needs to represent and manipulate the skeleton of an XML document for practical purposes[1].

The **XML Document data model** (**XD**) is composed of:

- a hierarchy of types (**XD-types**) that reflect the logical unit model;
- a set of functions (**XD-functions**) to manipulate the objects in the data model (**XD-objects**).
- a set of constants (**XDRE Toolkit**) that reflect the BNF-defs;

5.1 The XD-types

An XD-object belongs to one of the XD-types listed in appendix C, and represents an instance of a logical unit in the display.

In concrete terms an XD-object p is a list $(C\ s_1\ [s_2 \cdots])$ whose first element C is a symbol denoting the XD-type of p and whose other element(s) s_i may be compound objects of the tree generating from p (that is, p's **children**, which are XD-objects themselves) or a string which refers to the part of the user space connected with p.

An example of an XD-object of type 'attribute' may be:

[1] By the way, such a tool seems to be missing in the Emacs software world, so XD could actually be considered as the start of a useful tool.

```
(attribute (attName "length")
           (attValue "25.52cm"))
```

A 'seed' object is a special kind of 'element' object: it may have only three attributes in its header (namely "version", "encoding", "standalone"), does not have an element name and its children are limited as defined in appendix C.

5.2 The XD-functions

The XD data model provides functions for the manipulation of its objects. Their names start with `XD-` and follow these naming conventions:

- `XD-<...>` refers to a function that performs an operation on a child, for example `(XD-<get> etree 'header)` returns the entire object of type *header* of the object `etree`;
- `XD->...<` refers to a function that performs an operation on the contents of a child, for example `(XD->get< etree 'seed 'header 'eleName)` returns the string associated with the element name in the header of the seed of the object `etree`;
- `XD-{...}` refers to a function that returns a list of objects, for example `(XD-{getall} elt 'PI 'comment)` returns a list of all the comments and processing instructions contained in the element `elt`;
- `XD-...-p` indicates a predicate function, as for standard Lisp convention, i.e. a function that checks some condition and returns `nil` or `t`.

The XD-functions are documented internally in the code. A list is provided in appendix A for a general view and reference.

5.3 The XDRE toolkit

The **XDRE** toolkit is a set of string constants which are regular expressions that match some of the basic building blocks of XML, defined by the BNF-defs[2]. Their purpose is to be used in the parsing functions instead of literal regexps, for readability.

[2]Not all the BNF-defs can be translated into regular expressions, mostly because there is no trivial way of translating a BNF difference construct, such as in '(Char - ']')*', which indicates a sequence of zero or more instances of the BNF production 'Char' which are not ']'.

Each constant's name is of the form `XD-R-component` , where `component` reflects the name of a rule in the BNF-defs.

In constructing the regexp, the symbols '<<', '>>', '||', '**', '++', '--' are used in place of '\\(', '\\)', '\\|', '*', '+', '?' respectively.

The table in appendix B describes what each XDRE represents.

5.4 Structure of an Etree

The Etree is the structural representation of the file being edited. It is maintained and manipulated through the facilities provided by the XD data model.

The Etree is practically a 'seed' XD-object, that is, a list of objects which are lists themselves. A simple example of an Etree object may be:

```
(seed (header (eleName "")
              (attList (attribute (attName "version")
                                  (attValue "1.0"))
                       (attribute (attName "encoding")
                                  (attValue "UTF-8"))
                       (attribute (attName "standalone")
                                  (attValue "no"))
                       (attribute (attName "extDTD")
                                  (attValue "SYSTEM \"dtdfile.dtd\"")))))
      (comment "Simple document")
      (element (header (eleName "root")
                       (attList (attribute (attName "att1")
                                           (attValue "val1"))))
               (charData "This is some character data")
               (element (header (eleName "child")))
               (PI (PITarget "aTarget")
                   (PIBody "aBody"))))
```

This may be extracted from an XML document that looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE root SYSTEM "dtdfile.dtd">

<!-- Simple document-->

<root att1="val1">
This is some character data
   <child/>
   <?aTarget aBody?>
</root>
```

5.5 The Parser

The **XD Parser** (**XDP** for short) takes as input an Emacs buffer containing an XML document and extracts the relative Etree. The Parser checks the syntax of the document;

if the document cannot be parsed, it stops and point is left in front of the component that could not be parsed.

Emaxml parses an XML document by moving point in the buffer which contains the XML file. At the current position of point, the parser expects to find a sequence of characters that corresponds to one of a series of possible XD-object, according to the BNF-defs. If such a sequence is found, the relative XD-object is built (**object extraction**), and point is advanced, otherwise the parsing is unsuccessful.

The parsing process is a recursive one, so at the end of the day it consists of placing point at the beginning of the buffer and trying to extract a 'seed' object.

XDP consists of:

- a set of auxiliary functions (the **XDP Toolkit**), that carry out general operations related to parsing;
- a set of extracting functions (the **XD-PC-functions**), each of which is concerned with parsing an XML component.

These are described below.

5.5.1 XDP Toolkit

Parsing and in particular object extraction involve some elementary operations, provided by the XDP toolkit, that fall in one of the following categories:

- **Matching and skipping**

  The parser often needs to check if the text starting at point matches a particular regexp. It may need to retrieve it or ignore it. Functions like `XD-P-match-minus` or `XD-P-skip` provide such operations.

- **BNF construct handling**

  The objects to be extracted derive from the BNF-defs, which are composed of *conjuctions* (sequences), *disjunctions* (selections, '|') and *repetitions* ('*', '+', '?'). Functions in this category (such as `XDP-and` or `XDP-*`) provide these features.

- **Object manipulation**

  Functions in this category provide operations that are object-specific such as translating a standard entity reference to the corresponding character, or extracting in-

formation from the prolog of the XML document, or building an object from its components.

Generally speaking, XDP functions try to match the contents of the buffer at point with something (for example a regular expression or the result of one or more other XDP functions) and return what matched.

A return value of `t` means that the requested match was not found but the function is successful anyway. For example, when trying to match '0 or more instancies of something', a non-match is a success nonetheless.

All XDP functions are expected to leave point at the end of what they matched, or where it was if nothing was matched.

See Appendix D for the list and details of the XDP functions.

5.5.2 XD-PC parsing functions

Every XD-type has a corresponding XD-PC-function that parses the text at point and returns an object of that type if one was there, or `nil`.

Moreover, there are several XD-PC-functions that refer to some BNF-defs. The object returned by such a function is not in the XD data model, but is of the same structure of an XD-object. For example, `XD-PC-prolog` parses the prolog of an XML document as defined by the BNF-def number 22.

A return value of `nil` means that the object was not recognized at point.

Most of XD-PC functions are straight-forwardly constructed by reproducing the BNF-def using a combination of XDP functions, XDRE regexps and XD-PC functions themselves, e.g.:

```
[01] ;; [28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S?
[02] ;;                      ('[' (markupdecl | DeclSep)* ']' S?)? '>'
[03] ;; doctypedecl -> Name ExternalID? InternalID?
[04] (defun XD-PC-doctypedecl ()
[05]   (XD-P-build 'doctypedecl
[06]          (XD-P-skip "<!DOCTYPE" XD-R-S)
[07]          (XD-PC-Name)
[08]          (XD-P-01 (XD-P-and (XD-P-skip XD-R-S)
[09]                             (XD-PC-ExternalID)))
[10]          (XD-P-01 (XD-P-skip XD-R-S))
[11]          (XD-P-01 (XD-P-and (XD-P-skip "\\[")
[12]                             (XD-PC-InternalID)
[13]                             (XD-P-skip "\\]")
[14]                             (XD-P-01 (XD-P-skip XD-R-S))))
[15]          (XD-P-skip ">")))
```

The comment in lines 1-2 contain the BNF-def as from [1].

Line 3 describes what the object is composed of, i.e. a Name object, possibly an ExternalId object, possibly an InternalID object.

Line 5 invokes the XDP-build function to build a 'doctypedecl' object as described by lines 6-15.

Line 6 skips over '<!DOCTYPE' and whitespace.

Line 7 extracts a Name object.

Lines 8 and 9 deal with an optional pair, composed of some whitespace and an ExternalID object, and extract the latter.

Line 10 skips over some optional white space.

And so on.

5.5.3 Whitespace in parsing

The Parser is responsible of filtering the whitespace present in the XML file according to the chosen whitespace policy.

If the policy is "Allow-none", all boundary whitespace is removed from the character data.

If the policy is "Allow-all", all whitespace is preserved in the character data.

If the policy is "Allow-all-but-void", all whitespace is preserved, but void 'charData' objects are not.

5.6 The Writer

The Writer carries out the opposite of the Parser: it takes an Etree and produces an Emacs buffer whose contents are the XML document corresponding to that Etree.

The Etree received as input is assumed to be always errorless (i.e. to be formed of legal XD-objects as defined in appendix C). In the Emaxml context this is supposedly always true since it may only have been produced by the Parser or by the Emaxml mode, which both enforce syntactic control over the structure[3]

An XML document d is said to be **correctly produced** from an Etree e if and only if the result of parsing d is equal to e.

The function `XD-W-write` provides translation from an XD-object to an equivalent string in XML syntax.

In terms of whitespace, the Writer can produce an XML file optimized for:

[3]But see the footnote on page 19.

- Storage (i.e. with no extra whitespace added), if the policy is 'Allow-none'.
- Human inspection (i.e. the markup is indented), if the policy is 'Allow-all' or 'Allow-all-but-void'. The indentation style is implemented very simply at this stage of development, and can be improved later.

5.7 Whitespace handling

A piece of whitespace is a sequence of one or more spaces, tab characters, carriage return characters or linefeed characters. In the following discussion I refer to whitespace which is not part of markup, i.e. it is part of a piece of character data.

Whitespace which is between other non-whitespace characters is certainly part of the character data and must be preserved, while whitespace which is immediately before or after a piece of markup (from now on referred to as **boundary** whitespace), may be there for one of two reasons:

- Because it is integral part of the topic of the document (e.g. indentation such as in C code or poetry), and must be preserved.
- Because it is used to make the XML file more human-readable in raw XML format (e.g. blank lines, or tabs used for indentation). This whitespace does not affect the semantics of an XML file, and it should be up to the user whether to preserve it or not.

A sequence of whitespace characters only between two markup constructs corresponds in the Etree to a `charData` object whose string is whitespace only. Such an instance is called **void**.

In general, Emaxml is set to comply with one of the following policies for boundary whitespace, at the user's choice:

- **Allow-all**: do not perform any processing on the boundary whitespace.
- **Allow-none**: no boundary whitespace is preserved at all.
- **Allow-all-but-void**: preserve boundary whitespace but discard void `charData` objects. In practice, whitespace between markup that not contains any character data is considered to be for indentation purposes only. This is the default policy.

Chapter 6

The Mode

6.1 General description of an Emacs Mode

A *mode* is a set of definitions that customize Emacs and can be turned on and off by the user. There are two varieties of modes: *major modes*, which are mutually exclusive and used for editing particular kinds of text, and *minor modes*, which provide features that users can enable individually.

An example of a mode is 'C' mode, whose purpose is to edit C code files. This mode is activated when loading a C file, or by calling the Emacs Lisp function 'c-mode'. Some of the many features available when a buffer is in C mode are:

- the syntactic constructs of C are highlighted in different colors; this provides also instantaneous syntactic check;
- when the user types a closing brace the corresponding opening brace blinks;
- the text can automatically be indented according to one of many styles;
- the program can be compiled in Emacs by a keystroke;
- point can be moved to next/previous function;
- there are tools for version control and debugging.

Some of these functionalities are automatically managed by the mode, others are activated by a key sequence or by an item in a menu.

Most features of the mode can be finely tuned using Emacs's customization system.

6.1.1 Implementing an Emacs Mode

Implementing a mode consists basically of two phases[1]:

- writing the Lisp functions that perform the various operations;
- setting up the mode, that is, making Emacs aware of when and how to use those functions, and which way it should perform its common operations.

In general, a mode is set up by defining its components:

- The *keymap* maps key sequences to Lisp functions.
- The *syntax table* defines categories of characters in terms of the syntax;
- The *buffer-local variables* can be used to define the behavior of Emacs relative to some common functions. For example, C mode and Lisp mode both set the variable 'paragraph-start' to specify that only blank lines separate paragraphs. They do this by making the variable buffer-local in the buffer that is being put into C mode or Lisp mode, and then setting it to the new value for that mode.
- The *standard hooks* can be used to make Emacs perform some common operation in an appropriate way, peculiar to the new mode.
- *New hooks* are defined and set to default values. The new mode will call the functions listed in these hooks when performing particular operations, so allowing the user or a developer to customize the behavior of the mode by changing the values of the hooks.

6.2 Emaxml Mode

The Emaxml mode is based on the management of an Ebuffer and an Etree together. They are kept consistent with each other all the time, which means they always represent the same XML document.

The Ebuffer is an instance of an Emacs buffer, which is an internal data structure that Emacs is able to display. It contains therefore the information that relates the Etree to its visual representation.

The code of the implementation of Emaxml is commented internally. This section offers a general view and a discussion of the main issues related to the Ebuffer and its relationship with the Etree.

[1]Please refer to Appendix E for the definition of the technical terms in this section.

6.2.1 Ebuffer Issues

The two major characteristics of the Ebuffer are:

- It must carry information that relates its contents to the Etree.

  A logical unit is an entity of the Ebuffer, and has a logical correspondence with an XD-object, an entity of the Etree. This relationship must be somehow encoded in the Ebuffer.

- It is a *controlled buffer*, as defined in section 2.3.

  The control enforced by Emaxml over the Ebuffer is of two types:

  - Point movement control.

    In particular, point must be constrained to be always in user space and to move in a meaningful way in response to user commands.

  - Contents control.

    This type of control includes low-level syntactic and structural control (see section 3.9).

6.2.2 Ebuffer Implementation

The main idea behind the implementation of the Ebuffer is to use text properties to carry out control and overlays for a logical view of the buffer.

The Logics of the Ebuffer

An overlay is a data structure that is applied to a region. The region to which it applies is defined by the buffer to which the overlay belongs and the starting and ending buffer positions. Once an overlay has been created, it can be assigned values for a series of properties that will apply to all the characters covered by its region.

Some properties of an overlay are predefined and permit establishing the layout (e.g. the 'face' property) and the behavior (e.g. the 'intangible' or the 'modification-hooks' properties) of the covered characters. But, a property can also be "invented" and a value assigned to it for an overlay.

The function 'emaxml-insert-object' takes an XD-object of any XD-type[2] and inserts it in the Ebuffer at point. It also creates an overlay to cover the region occupied by that object. For example, at the activation of the mode 'emaxml-insert-object' takes the entire Etree and recursively inserts the whole of it in the Ebuffer, creating an overlay for each subtree. The result is a structure of overlays that contain each other in the same way the subtrees in the Etree do. Each overlay gets assigned the XD-object that generated it as the value of its 'subtree' properties.

Hence, an overlay *is* a logical unit, and the corresponding subtree is connected directly: the three concepts can be used interchangeably. This mechanism provides a straightforward mapping between a buffer position p and the set of subtrees S it belongs to: S is defined as the set of values connected to the 'subtree' property of all overlays covering p, which form the set O.

O can be retrieved with the appropriate Emacs functions. It will be in no particular order. Then it can be ordered by **specificity** with respect to p according to the starting and ending positions of the overlays[3]. The result is the **ordered list of overlays** Ω covering p, from the most specific up to the Seed.

The **ordered list of subtrees** Σ describes the way from the elementary subtree to which p belongs up to the 'seed' XD-object.

From Ω and Σ a number of useful informations are acquired about p. A set of functions that extract them are provided in the code of Emaxml, and allow the level of abstraction to be lifted from reasoning about overlays and buffer positions to reasoning about logical units, parental relationships, XD-types, containing branches, etc. This set can easily be extended. Here are a few examples:

- The elementary logical unit (or simply the logical unit) at p is the first element of Ω.
- The type of the logical unit at p is the type of the first element of Σ.
- The parent of the subtree at p is the second element of Σ.

[2]It takes also another few parameters that are not mentioned here because not strictly related to this discussion.

[3]To disambiguate the cases when two or more overlays cover the same region, a **priority** property is also assigned to the overlays, with the elementary logical units having the highest priority, and the Seed the lowest.

- The branch at p is the first element of Ω of a primary type.

- The string corresponding to the contents of the logical unit at p is the buffer substring from the start to the end of the first overlay in Ω if p is in a monoline logical unit, or that minus the non-user space if p is in a multiline logical unit.

This type of functions form the **Ebuffer/EtreeConnectivity Toolkit**.

An overlay also carries more information about the logical unit it represents, such as its depth in the tree, the type of space ('user' if the logical unit is elementary, `nil` if it is compound), etc. When an error is to be signaled about a compound logical unit, the error face is attached to its overlay (generally changing its background color), and the value of its 'default-face' property restored when the error is corrected.

When p = point, the adjective **current** can be added to the various entities and properties related to p, such as in "the current logical unit", "the current branch", "the current type", "the current depth", etc.

Space Implementation

One of the tasks needed to implement the control features of the Ebuffer is to implement the Emaxml space model as described in section 2.3.

To make a buffer cell automatic, it is basically assigned the text properties 'intangible' and 'read-only'. Moreover, it is made non-sticky in both directions to avoid inheritance when inserting text in the adjacent cells.

Semiautomatic characters exploit a particularity of the 'intangible' property: point is not allowed to be between two characters that have the *same value* for the 'intangible' property. Let us consider a portion of the buffer as in figure 6.1.

The buffer cells from $x+1$ to $x+5$ must be made automatic, and the cell between x and $x+1$ semiautomatic. So the value q is assigned to the 'intangible' property from x to $x+5$, and different values are assigned to the adjacent characters. Point will not be allowed to be between characters with the same value for the 'intangible' property, so the desired cells will realize the desired type of space.

A semiautomatic cell is also 'read-only' and 'non-sticky' in both directions.

The entire buffer can be viewed as a large area of automatic space, with "holes" of user space delimited at the right end by one semiautomatic cell. The values for the 'intangible' property must reflect this. A new value is generated every time a new semiautomatic character is inserted, and used by the following automatic characters. When any portion of

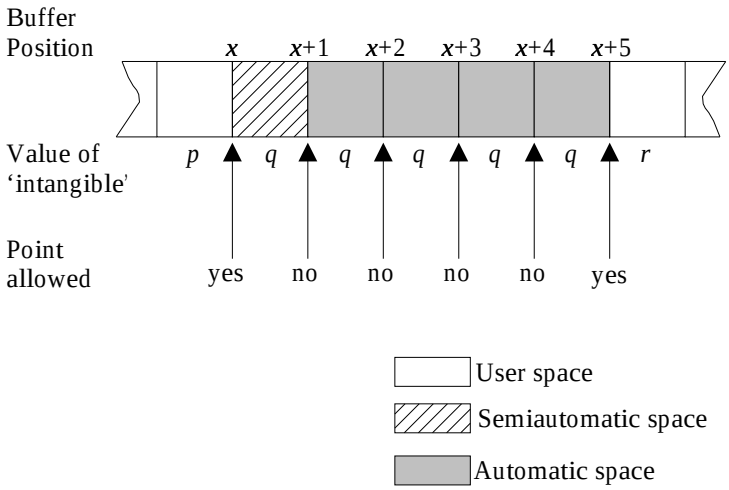

Figure 6.1: Portion of a buffer.

buffer that contains mixed space (i.e. a logical unit) is inserted or deleted, the buffer must be kept consistent by merging the appropriate sequences of automatic characters, that is, by ensuring they have the same value of the 'intangible' property as appropriate.

Movement Control Implementation

Most of the movement control is taken care of by Emacs once the 'intangible' properties have been set. The cursor will move only in user space. However, the Emacs response to the vertical movement is not satisfactory and has been rewritten.

Also, Emaxml takes care of placing point after a backward deletion at the beginning of an elementary logical unit, and when point is moved to the end or beginning of the line.

Due to the characteristics of the 'intangible' property described earlier, it is not possible to make the very first character of the buffer be forbidden to the cursor. In fact, position 1 cannot be between two characters with the same value for the property, because it has no character on the left. The same applies to the very last position of the buffer. To obviate this problem, a function that checks these conditions and move point to a suitable place if needed is hooked to the 'post-command-hook' variable, whose contents is evaluated every time the command cycle of Emacs comes to an end.

Contents Control

Both syntactic and structural control are realized by a function hooked to the 'after-change-functions' variable.

This function, amongst other tasks, checks if the latest change has made any of the following conditions true, and possibly calls the function that manages errors:

- The current string does not parse.
- An 'attName' is empty but the corresponding 'attValue' is not.
- The Root Name in the Document Type Declaration is empty but the rest of the Document Type Declaration is not.
- An Element Name is empty.

As part of the contents control, the Element Name of the root element and the Root Name are always kept equal.

Chapter 7

Performance assessment

Emaxml has been used extensively and has informally proved to meet the functional requirements described in this document, but it should still be tested more formally on this.

Two requirements that it should meet are:

- The correctness of the XDP parser and the XDW writer, on which Emaxml relies heavily.
- The consistency of the Etree and the Ebuffer with each other, at any instant.

To prove these two characteristics of Emaxml, the tests in the following two sections have been devised and succesfully carried out to some extent.

However, for Emaxml to be considered reliable more testing should be designed.

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 test are those described in the Emaxml specification in part II.

7.1 XML-equivalence of two documents

The correctness of the parser and the writer are proved by parsing a file then writing it back. The original and the produced files should be equivalent. The *canonical form* of an

XML document is the measure of equivalence used: if two files have the same canonical form, they represent the same XML document. This is established and documented by the World Wide Web Consortium at http://www.w3.org/TR/xml-c14n.

To produce the canonical form of a file, the Java class `'jd.xml.xslt.Stylesheet'` is used. This is part of the XSLT processor `'jd.xslt'` from Johannes Döbler, the master copy of which is at http://www.aztecrider.com/xslt/jd.zip.

In more formal terms:

- Let us consider an XML document in file d_0.
- The parser parses d_0 and produces its Etree e_0.
- The writer writes e_0 in a file, saved as d_1.
- d_1 is canonicalized into d'_1.
- d_0 is canonicalized into d'_0.
- d'_1 and d'_0 are compared.

If d'_1 and d'_0 are equal, the parser and the writer are correct.

The code for this test is listed as function `'test02'` in appendix G.

Emaxml has passed this test on the test files listed in appendix F.

7.2 Consistency of the Emaxml Mode

Emaxml works on the assumption that the Etree and the Ebuffer represent always the same XML document.

The Etree is the `'subtree'` property of the overlay corresponding to the Seed. If this XD-object is written into a buffer and then Emaxml mode is activated, the two buffers should have exactly the same contents. In particular, they should have the same characters with the same text properties and an equal set of overlays covering the same regions and connected to equal XD-objects.

This test effectively proves that the structure of the overlays is in order, which is a condition that cannot be proved visually.

The test can be run in "interactive" or "batch" mode. In batch mode it only displays a message with the result of the test. In interactive mode it also displays the two buffers and

a buffer with a brief report. If any difference is encountered, the first character to differ in both buffer is highlighted.

Note that this test has no meaning if there are erroneous logical units in the original buffer, because they will not be reproduced with the same properties. It should be improved to allow all cases, because it is a very powerful debugging tool to test new features and see if they keep the Ebuffer in order.

All the implemented features of Emaxml that change the Ebuffer have been tested extensively in this way.

The code for this test is listed as function `'test03'` in appendix G.

Part IV

Future

Chapter 8

The Future of Emaxml

"What is it?"

"The ... uh... stuff that dreams are made of."

Sam Spade in *The Maltese Falcon*

Emaxml is not a finished product.

The idea behind it has proved to be good enough to interest the people I have talked to who use Emacs for editing XML. In fact, there is a need in the Emacs software world for a better mode for editing XML. But Emaxml must be improved seriously to be able to fill this gap.

Emaxml will be distributed as an open source project, and may arouse the interest of programmers around the world. I have no experience of open source projects, so I think I will start with laying down the ideas that have come about in the discussions with my supervisor and the people I have showed the present Emaxml version. Then I will send messages to some appropriate mailing lists and see what happens.

A good starting point may be Savannah (http://savannah.gnu.org/), a facility of the GNU project for development, distribution and maintainance of Free Software. It allows contributors to easily join existing Free Software projects.

I think that the moral issues behind the Free Software idea are a matter of personal belief, but it is out of discussion that the results obtained by developing in the open source paradigm have proved to be incomparable to those obtained by people who get paid for programming, and surely are hugely better than those obtained by working alone, because of the synergy.

In the following sections I describe ways in which Emaxml could be improved.

8.1 Enhancing the Low-Level Emacs Functions

Many editing operations are implemented by writing new functions for old functionalities. Instead, the low-level functions that already do a similar operation should be enhanced to handle Emaxml buffers properly. That way, all of the other functions in Emacs that are built on top of them would automatically be extended to work for Emaxml.

For example, the kill-ring management is based on a few low-level functions. If these are enhanced, all the other more sophisticated functions for yanking and killing will be available directly, because they are based on the low-level ones.

8.2 DTD awareness

Emaxml could benefit from becoming DTD-aware. This would give it the ability to enforce high-level control over the contents of the document. An editor with such capability can help the author in many ways.

For example:

- When a new element is added, it can be created as a skeleton, with the compulsory children and with default values for the attributes.
- Addition and deletion of attributes can be controlled for validity.

8.3 Improvements to the Existing Features

8.3.1 Activating the Mode

It should be possible to have Emaxml as the default mode for editing XML if the user wants to.

8.3.2 Point Movement

Commands may be implemented for moving by branches, and bound for example to the existing commands for moving by sexp[1].

8.3.3 Elementary Editing

Two consecutive Character Data logical units are presently permitted. They should be forbidden, since such a thing has the same meaning as one such logical unit with the

[1] A sexp, as far as moving is concerned, is a piece of text enclosed in brackets.

combined contents. The place for checking this is in the addition functions.

8.3.4 Killing and Yanking

Killing and yanking should be extended to handle complete and incomplete subtrees instead of strings only. This is discussed in section 8.4.3

Commands to kill the following or preceding branch may be implemented, to parallel `'kill-sexp'`. To feel "natural" (and consistent with how it works in the rest of Emacs) the cursor should be positioned "before" or "after" a branch. Such concepts are not clear in Emaxml, but the use of the separator as described in 8.4.2 would make them possible.

8.3.5 Adding Branches

Adding branches is one of the commonest task in editing. The way it is implemented now is very poor and unnatural. A discussion on how to improve it is in section 8.4.2

8.3.6 Error Management

Presently Emaxml records only one error per logical unit. Moreover, if an elementary logical unit *and* one or more of the containing logical units have errors attached to them, only the error connected to the elementary logical unit will be displayed when requested.

This could be improved by having a list of errors attached to a logical unit, and by displaying all the error messages related to any logical unit point is in.

Another issue is that presently Emaxml does not check for parse errors in the Internal DTD. Parse errors are detected using the parsing functions of the XDP parser when a change to the Ebuffer occurs. For example if a change occurs in an Attribute Value, `'XD-PC-attvalue'` is invoked by the function `'emaxml-current-parse'` to verify the validity of the new value. For the Internal DTD there is not such a function, which should be written and then used in `'emaxml-current-parse'`.

8.3.7 Saving

Emaxml saves the document even if it contains errors.

This should be avoided, for example by advising the user that errors are present in the document and asking them whether they want to save it anyway.

8.4 New Features

Some of the ideas described below were part of the project since its beginning, others came about during the development or from discussion with my supervisor and other people involved in editing XML with Emacs.

8.4.1 Display Modes

A branch could 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).

These characteristics are independent so there are four ways of displaying a subtree (see Fig.8.1), called **display modes**: outline-expanded, outline-collapsed, inline-expanded, inline-collapsed.

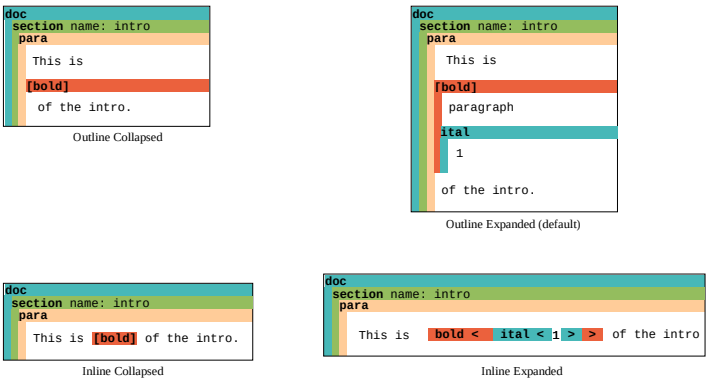

Figure 8.1: Display modes

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.

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 or in an additional file) and restored the next time the document is visited in Emaxml mode.

8.4.2 Zero-Length Character Data logical units between branches.

Presently, to add a new branch the user has to place the cursor on a branch and press a key combination to add the new branch as a child or sibling of the current branch. This is a little cumbersome, especially when writing text-related documents.

A more natural way of doing this is to change the XD model to have "length zero" (or, simply, empty) `'charData'` objects between any two non-`'charData'` objects. This extension to the model would make it more versatile. It would also reflect better what the reality is: there is a (possibly zero-length) sequence of character data between any two pieces of markup.

In practice, a Character Data would be displayed as before except when it is empty. In this case it would be displayed as the "separator", the blank line that presently divides the branches. The user may just place the cursor there and start writing, and a new Character Data would be created.

A further improvement may be that Emaxml recognizes some sequences of characters typed in an empty Character Data and adds an appropriate logical unit instead of always a Character Data.

For example if the user types '<', Emaxml waits for the next character. Then:

- If it is '<' again (or a character which would be illegal as the first character of an Element Name), the user meant to insert one greater-than sign (or one greater-than sign plus the second character) at the beginning of a new Character Data. So a new Character Data is added with those characters at the beginning.

- If it is a legal character for the start of an Element Name, the user meant to add a new Element whose Element Name starts with that character, and that is what is added.

8.4.3 A New Meaning for the Region

A region is the portion of a buffer included between two buffer positions. In particular, *the* region in delimited by point and the mark.

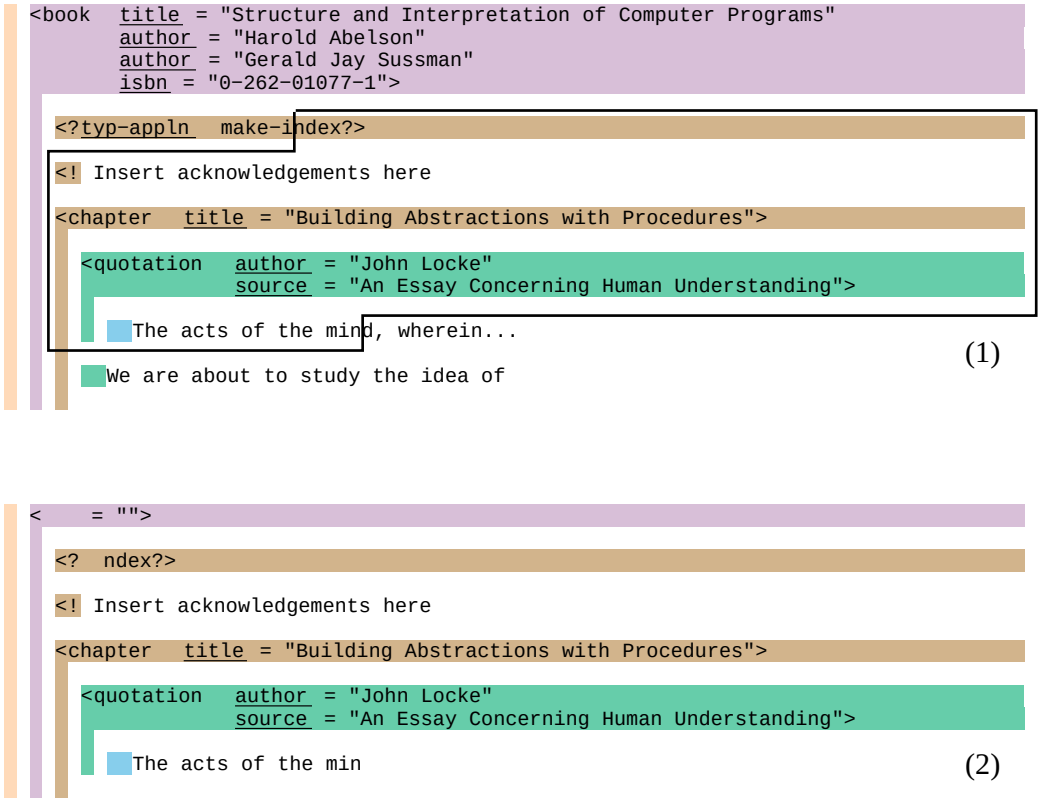

Figure 8.2: A possible interpretation of an incomplete region. (1) The region (2) Its meaning.

In a normal buffer this sums up to a string, but in an Ebuffer it would represent a subtree.

A subtree is said to be **complete** if it is an entire logical unit. Then it corresponds to a well-formed XD-object and can be somehow saved and retrieved as such.

An **incomplete subtree**, on the other hand, corresponds to a region whose boundary positions are not those of an entire logical unit[2]. A possible interpretation of such a region in terms of the Etree is that it should be the minimum logical unit containing the region completely, but without its children completely excluded from the region. Moreover, empty logical units should be added to this tree where needed to make it complete.

An example is shown in figure 8.2.

So a region could be stored as an XD object.

[2]For example, a region that begins in the middle of a Element Name of an Element and finishes somewhere in another Element.

8.4.4 Killing and Yanking Extended

Killing and yanking operations manipulate strings only. If the region as defined in the previous section is implemented, incomplete subtrees could be killed and stored as complete subtrees with the necessary empty logical units.

The meaning of yanking a subtree must be investigated. For example yanking a Element inside a Processing Instruction Target could be considered either invalid or meaning to yank the Element as a sibling of the Processing Instruction.

Moreover, it may be very useful to be able to kill a portion of an Emaxml buffer and yank it as XML in a non-Emaxml buffer, such as an email message, or viceversa. The internal format for this sort of transformation would naturally be XD, then the stored tree should be translated into the appropriate format for the target buffer.

8.4.5 Undo

Undoing is one of the most useful facilities of an editor. It is not implemented now and it should be one of the first priorities.

The undo list in Emacs is a list whose elements are insertions or deletions. From the current status of a buffer, the previous one can be re-established by "undoing" the action described in the front element of the undo list. The operations that can be saved unfortunately are only text insertion/deletion, point movement, marker movement and text properties changes.

A complex operation such as the deletion of a branch cannot be described in these terms. Moreover, there are no variables to which one could hook a function to be called before undoing. So the standard undo list does not seem a valuable option.

"Advice" could be used here to force the undo functions to do something specific if the front of the undo list is the description of an Emaxml operation.

Undo should at least be implemented as constrained to operate on the current elementary logical unit. This ability already exists in Emacs, which has the feature that when Transient Mark mode is on, undo is constrained to the region.

8.4.6 Miscellaneous Ideas

- There is a little bug in Emaxml display. The second and subsequent Attributes of an Attribute List are slightly dis-aligned with respect to the first one. This is due to

the fact that the Element Name is "boxed". This should be fixed. One solution is to eliminate the box. Another may be to put a box the same colour of the background around (part of) the `'attsidebar'` of the offending Attributes.

- It may be useful to have commands to swap two branches, the way `'transpose-words'` works.
- A nice feature would be selective spell checking, limited perhaps to character data only.

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.

[6] H. Abelson, G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

An inspiring book, accidentally about Lisp, and purposefully about abstraction. It has been said that its footnotes alone are more interesting than most books around.

[7] M. Gankarz. *The UNIX Philosophy*. Digital Press, 1995.

Software Engineering is not just a waterfall.

Appendix A

Details of XD Functions

XD-<get> (obj &rest fields)

Retrieve an object's descendant of a certain type.

XD-<get-a> (obj field)

Retrieve the first of an object's descendants that is of a certain type.

XD-<get-or-empty> (obj &rest fields)

Retrieve an object's child of a certain type, or an empty one if one does not exist.

XD->get< (obj &rest fields)

Retrieve the value of an object's descentdant of a certain type.

XD->get-a< (obj field)

Retrieve the value of the first of an object's descendants that is of a certain type.

XD->set< (obj str)

Set (destructively) the value associated with elementary object OBJ to STR.

XD-{getall} (obj &rest field)

Return a list of all sons of OBJ's which are of one of the types FIELDS.

XD-<copy> (x)

Return a copy of of an object.

XD-<empty> (type)

Return an empty object of an XD-type.

XD-<index> (obj son)

Return the position a son is in the list of children of an object.

XD-<insert-after> (obj old-son new-son)

Insert (destructively) NEW-SON after OLD-SON in OBJ.

XD-<insert-last> (obj son)

Insert (destructively) SON as last son of OBJ. Return SON.

XD-<insert-nth> (obj son n)

Insert (destructively) SON as Nth son of OBJ. Return SON.

XD-<remove> (obj son)

Remove SON from the children of OBJ.

XD-elementary-obj-p (obj)

Return `t` if OBJ is an elementary XD-object, `nil` otherwhise.

XD-empty-p (obj)

Return `t` if OBJ is an empty object, `nil` otherwhise.

XD-obj-p (obj)

Return `t` if OBJ is an XD-object, `nil` otherwhise.

XD-oftype-p (obj type)

Return TYPE if OBJ is of type TYPE or `nil` if it's not.

XD-oftypes-p (obj &rest types)

Return the type of OBJ if OBJ is of a type in TYPES.

XD-primary-type-p (type)

Return `t` if TYPE is a primary XD-type.

Appendix B

Details of XDRE constants

BNF-def	XDRE constant	Explanation
[2] Char	XDRE-Char	Unicode character range
[3] S	XDRE-S	White Space
[87] CombiningChar	XDRE-CombiningChar	Among others, this class contains most diacritics
[89] Extender	XDRE-Extender	Extenders
[85] BaseChar	XDRE-BaseChar	Among others, this class contains the Unicode alphabetic characters of the Latin alphabet
[86] Ideographic	XDRE-Ideographic	Unicode ideographic characters
[84] Letter	XDRE-Letter	BaseChar's + ideographic characters
[88] Digit	XDRE-Digit	Unicode digits
[4] NameChar	XDRE-NameChar	Characters allowed in Names
[5] Name	XDRE-Name	Matches a legal Name
[25] Eq	XDRE-Eq	Equality sign
[68] EntityRef	XDRE-EntityRef	Matches an entity reference (eg. '&cright;')
[66] CharRef	XDRE-CharRef	Matches a character reference (eg. 'Ћ')
[19] CDStart	XDRE-CDStart	Matches '<![CDATA['
[21] CDEnd	XDRE-CDEnd	Matches ']]>', the CDATA section terminator
[69] PEReference	XDRE-PEReference	Matches a Parameter Entity (eg. '%abc;')
[26] VersionNum	XDRE-VersionNum	Matches the version number declaration in an Xml Declaration
[81] EncName	XDRE-EncName	Matches the encoding name in an Encoding Declaration
[13] PubidChar	XDRE-PubidChar	Characters allowed in names of PubidLiteral's

Table B.1: The XDRE toolkit

Appendix C

Details of the XD-types

Below is the list of the types belonging to the XD data model.

The notation used is:

→: "has as children";

*: "zero or more";

?: "zero or one";

+: "one or more";

|: indicates alternative children.

seed	→	header doctypedecl? (comment | PI)* element (comment | PI)*
doctypedecl	→	Name ExternalID InternalID
Name	→	*string*
ExternalID	→	PubidLiteral? SystemLiteral?
PubidLiteral	→	*string*
SystemLiteral	→	*string*
InternalID	→	*string*
element	→	header (element | comment | PI | entRef | charRef | charData)*
header	→	eleName attList*
eleName	→	*string*
attList	→	attribute+
attribute	→	attName attValue
attName	→	*string*
attValue	→	(*string* | entRef | charRef)*
comment	→	*string*
PI	→	PITarget PIBody
PITarget	→	*string*
PIBody	→	*string*
entRef	→	*string*
charData	→	*string*

Table C.1: Data types in the XD data model.

Appendix D

Details of XDP functions

XD-P-* (form)

Checks if point is at 0 or more occurrencies of FORM.

XD-P-01 (form)

Checks if point is at 0 or 1 occurrencies of FORM.

XD-P-match (re)

Checks if point is looking-at RE.

XD-P-match-minus (re1 re2)

Checks if point is looking-at the difference regexp (RE1 - RE2).

XD-P-match-until (re1 terminator)

If looking-at 'RE1*TERMINATOR' return what matches 'RE1*' and set point at end of it.

XD-P-skip (re)

Skips over a regular expression. Used for portions of buffer that don't represent any object.

XD-P-build (items)

Constructs an object by putting together the results of the forms in ITEMS. It is based on a call to XD-P-and whose result is put in a one-element list.

XD-P-and (forms)

Handles sequences. It also deals with forms that return `t` to mean a successful non-match, by not appending the `t` to the list returned.

XD-P-or (forms)

Handles selections.

Appendix E

Glossary of Emacs technologies

This chapter briefly defines some technologies related to the design of Emaxml. It is not intended to be exhaustive. For a complete and better explanation refer to the Emacs manual ([4]).

Face A face in Emacs jargon is a set of layout attributes, namely: font family, width, height, weight, slant, underline, overline, strike-through, box, inverse-video, foreground, background, stipple, inherit.

Echo Area The Echo Area is a line at the bottom of an Emacs frame, for displaying messages.

Keymap The keymap is the data structure that records the bindings of key sequences to the commands that they run. For example, the global keymap binds the character 'Ctrl-n' to the command function 'next-line', therefore when 'Ctrl-n' is pressed the cursor moves to the next line.

One of the characteristics of an Emacs mode is which key combinations trigger which operations. These are defined by the mode keymap.

Syntax Table A syntax table provides Emacs with the information that determines the syntactic use of each character in a buffer. This information is used by the parsing commands, the complex movement commands, and others to determine where words, symbols, and other syntactic constructs begin and end.

Each buffer has its own major mode, and each major mode has its own idea of the syntactic class of various characters. For example, in Lisp mode, the character ';' begins a comment, but in C mode, it terminates a statement. To support these

variations, Emacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode.

Hook A hook is a variable where it is possible to store a function or functions to be called on a particular occasion by an existing program.

For instance, if the name of a function is added to the variable 'after-save-hook' (using function 'add-hook'), that function will be called after saving any file.

Appendix F

Test Cases

The following files have been used to test the various features of Emaxml:

- `'bookcase.xml'`

  An XML file used as example in the [2]. It is supposed to cover most if not all the features of XML. The other files related to it are:

  - `'Bookcase_ex.ent'`
  - `'furniture.dtd'`
  - `'parts_list.ent'`

  They are all in the floppy, in `'xml/nutsample'`.

- `'imagelib.xml'`

  A short file that can be normally found in any distribution of Linux.

  The DTD is in `'imagelib.dtd'`. They are both on the floppy in `'xml/imagelib'`.

Appendix G

Test Code

The following is the code for the XML-equivalence test.

```
(defun test02 (d0)

  "Test Parser+Writer correctness through canonicalization.

Description:
Given the file name of an XML document d0, produce its etree e0 then
write d1 from e0. Canonicalize d0 and d1 and compare them with
diff. Return t if d1 and d0 are canonically equivalent.

   XDP     XDW     CAN
d0 ---> e0 ---> d1 ---> d1'--|
                             | diff
   CAN                       |------> equivalent?
d0 ---> d0'------------------|

Usage:
 Load emaxml.el, containing the data model and its
 components.
 M-x test02 RET.
 Give file name of the XML document.
"

  (interactive "FXML file:")

  (save-window-excursion
    (if (equal d0 "#")
        (setq d0 "~ceepd1/tesi/xml/nutsample/bookcase.xml"))

    (let (e0
          xdw-buf
          (d1 (concat d0 ".d1"))
          d0prime-buf
          d1prime-buf
          d0prime-str
          d1prime-str)

      ;; parse

      (setq XD-whitespace-policy "all")
      (message "Parsing...")
      (setq e0 (XD-parse d0))
```

```
      (if (null e0)
          (error (concat "test02: Error parsing " d0)))

      ;; write

      (setq XD-whitespace-policy "none")
      (message "Writing...")
      (setq xdw-buf (XD-write e0 nil))

      ;; save d1

      (write-file d1)
      (kill-this-buffer)

      ;; canonicalize

      (setq d0prime-buf (generate-new-buffer "d0prime"))
      (setq d1prime-buf (generate-new-buffer "d1prime"))

      (message "Canonicalizing the original document...")
      (shell-command (concat "java jd.xml.xslt.Stylesheet -out:method canonical-xml "
                             d0 " {identity}")
                     d0prime-buf)

      (message "Canonicalizing the produced document...")
      (shell-command (concat "java jd.xml.xslt.Stylesheet -out:method canonical-xml "
                             d1 " {identity}")
                     d1prime-buf)

      ;; compare

      (set-buffer d0prime-buf)
      (setq d0prime-str (buffer-substring (point-min) (point-max)))
      (set-buffer d1prime-buf)
      (setq d1prime-str (buffer-substring (point-min) (point-max)))

      (if (not (string= d0prime-str d1prime-str))
          (message "Test unsuccessful. :-(")

        (message "BINGO. The two canonical forms are equivalent.")
        (delete-file d1)
        (kill-buffer d0prime-buf)
        (kill-buffer d1prime-buf)))))



(defun test03 (batch)
  "Test consistency of an Emaxml buffer.

An Emaxml buffer is consistent if the etree and the ebuffer reflect
the same structure. Moreover, all the overlays must cover the area
corresponding to their subtrees.

To prove consistency of an ebuffer b1, a temporary ebuffer b2 is
produced as follows:

- the etree e1 from b1 is written with the XD-Writer to b2;
- Emaxml mode is activated in b2, making it an ebuffer.

Buffers b1 and b2 must be equal, and the sets of their overlays must
be equal.
```

```
This test is useful if applied to an ebuffe that has been modified.

BATCH, if non-nil, indicates not to output the results in windows.

Return nil for error, t for OK."

  (interactive)

  (let* ((e1 (emaxml-whole-etree))
         (b1 (current-buffer))
         (l1 (point-max))
         (ovs1 (overlays-in (point-min) (point-max)))
         (res-buf (and (kill-buffer (get-buffer-create "test03-result"))
                       (get-buffer-create "test03-result")))
         (b2 (and (kill-buffer (get-buffer-create "test03-b2"))
                  (set-buffer (get-buffer-create "test03-b2"))))
         l2
         ovs2
         (p 1) ;; position being examined
         maxp
         c ;; character being examined
         (inhibit-point-motion-hooks t)
         (inhibit-read-only t)
         (inhibit-modification-hooks t)
         contents-differ
         first-contents-diff
         overlays-differ
         (XD-whitespace-policy "none")
         success)

    ;; produce b2
    (message "test03: writing...")
    (let ((XD-whitespace-policy "but"))
      (XD-W-write e1 0))
    (message "test03: parsing & displaying...")
    (emaxml-mode)
    (setq l2 (point-max))
    (setq ovs2 (overlays-in (point-min) (point-max)))
    (setq maxp (1- (min l1 l2)))


    ;; compare lengths
    (with-current-buffer res-buf
      (insert (format "Length of b1: %d\n" l1))
      (insert (format "Length of b2: %d\n\n" l2)))

    ;; compare buffer contents
    (message "test03: comparing contents...")
    (while (<= p maxp)
      (with-current-buffer b1
        (setq c (buffer-substring p (1+ p))))
      (with-current-buffer b2
        (unless (string= c
                         (buffer-substring p (1+ p)))
          (put-text-property p (1+ p) 'face 'isearch)
          (setq contents-differ t)
          (unless first-contents-diff
            (setq first-contents-diff p))))
      (setq p (1+ p)))

    (with-current-buffer res-buf
      (insert "The contents are\n"
```

```
          (if contents-differ
              (concat "DIFFERENT.\nFirst difference:"
                      (number-to-string first-contents-diff))
              "EQUAL.\n\n")))

;; compare overlays
(message "test03: comparing overlays...")

(let ((ovs1-list (test03-ovs-to-ovs-list ovs1))
      (ovs2-list (test03-ovs-to-ovs-list ovs2)))

  (dolist (ov ovs1-list)
    (when (member ov ovs2-list)
      (setq ovs1-list (delete ov ovs1-list))
      (setq ovs2-list (delete ov ovs2-list))))

  (when ovs1-list
    (test03-color-overlays b1 ovs1-list)
    (setq overlays-differ))
  (when ovs2-list
    (test03-color-overlays b2 ovs2-list)
    (setq overlays-differ)))

(setq success (not (or contents-differ
                       overlays-differ)))

(with-current-buffer res-buf
  (insert  "The overlays are\n"
           (if overlays-differ
               "DIFFERENT.\n\n"
             "EQUAL.\n\n")
           (if success
               "BINGO."
             "The test didn't succeed.")))


;; display windows
(unless batch
  (message "")
  (switch-to-buffer res-buf)
  (delete-other-windows)
  (split-window nil 30 'horiz)
  (switch-to-buffer-other-window b1)
  (if (null first-contents-diff)
      (goto-char (point-min))
    (goto-char first-contents-diff)
    (recenter 3))
  (split-window)
  (switch-to-buffer-other-window b2)
  (if (null first-contents-diff)
      (goto-char (point-min))
    (goto-char first-contents-diff)
    (recenter 3)))

;; display message
(if batch
    (message "test03: Result -- Contents:%s Overlays: %s -- %s"
             (if contents-differ
                 "NOT OK"
               "OK")
             (if overlays-differ
                 "NOT OK"
```

```
                  "OK")
                (if success
                    "BINGO."
                  "SORRY.")))
    success))



(defun test03-ovs-to-ovs-list (ovs)
  "Return a list of lists, each of the form (START END SUBTREE)."
  (let (res)
    (dolist (ov ovs)
      (append (list (list (overlay-start ov)
                          (overlay-end ov)
                          (overlay-get ov 'subtree)))
              res))
    res))

(defun test03-color-overlays (buf lst)

  (dolist (ov-l lst)
    (overlay-put (make-overlay (car ov-l)
                               (cadr ov-l)
                               buf)
                 'face 'emaxml-face-error)))



(defun test03-batch () (interactive) (test03 t))
(defun test03-interactive () (interactive) (test03 nil))

(global-set-key "\C-ct3b" 'test03-batch)
(global-set-key "\C-ct3i" 'test03-interactive)
```

Appendix H

The Contents of the floppy

The floppy attached to the Dissertation Document contains the following directories and files.

```
.
|-- doc                                 Documentation
|   `-- diss.pdf                        This document, in pdf format
|
|-- misc-ref                            Reference
|   `-- grammar                         Grammar specifications
|       |-- bnfs                        List of BNF in w3c-xml-rec
|       |-- namespaces.htm              Document on namespaces
|       |-- unicode.htm                 Document on unicode
|       |-- w3c-xml-rec.htm             XML specification in html format
|       `-- w3c-xml-rex.txt             XML specification in ASCII
|
|-- prg                                 Code directory
|   |-- emaxml.el                       XD+emaxml code
|   `-- tests.el                        Test code
|
`-- xml                                 Sample XML code
    |-- book.xml                        The file used throughout this document
    |-- cs4.xml                         A long XML file
    |-- imagelib                        An XML document
    |   |-- imagelib.dtd
    |   `-- imagelib.xml
    `-- nutsample                       An XML document from [2]
        |-- Bookcase_ex.ent
        |-- bookcase.xml
        |-- furniture.dtd
        `-- parts_list.ent
```