m4x Internals

Diogo Kollross

June 19^{th}, 2001

Contents

Chapter 1

Introduction

1.1 Overview

This document will be a guide to everyone who wants to have a more depth view of m4x. Here you will find all about the design of m4x, and details of its implementation.

If you find any incorrections or misspelling, please send me an email at `kollross@users.sourceforge.net`. My knowledge of the English language is far from perfect. All comments contribute to the development of this document.

1.2 Copyright

Chapter 2

Analisis

2.1 Historic

The job of creating and maintaining a Web page, especially when writting by hand, is often awful and extremely tedious. Whenever there is a change in the layout of the pages, all files must be edited, one by one. Want to change the color of the titles from blue to black? Open each one of the (sometimes hundreds of) HTML files and change those two simple digits in the color attribute. The examples could go on forever. There must be and easier way.

What if one could split content from presentation?[1] What if just one command updated all files at once, putting the days of tedious editing in a dark past to be forgotten?

That is why I started writting m4x. Its name derives from m4, the macro processor and means *Macro For XML*. Mainly because lack of planning, this program was rewritten several times.

The first version was started by the end of 2000 and was being developed in C++ with the GLIB library. I choose C++ because I wanted to learn object oriented programming.

Suddenly, I decided that portability was one of the goals and abandoned GLIB.[2] I was using many of the features of GLIB, so I started rewritting the classes and duplicating the funcionality of GLIB. I even created one of (I believe) most rewritten code: a string class with operator overriding.

At this point I realized that, after some months of work, I had nothing. Assuming that the problem was the language, I decided to restart everything, now in C. This is the version where I went farthest. I created a stub program that recognized the command line options. I had a parser to the template files. I even had a working XML parser!

[1] Actually HTML purpose used to be splitting content from presentation, but through the years it was extended to fulfill the needs of the commercial Web and became the "Frankenstein" whe know today.

[2] I can´t understand it either, as GLIB was created to be portable...

Then I got stuck. Not that I didn´t know what to do next, but I realized that “that” was not what I wanted. The code was a mess, the XML parser worked but was almost unintelligible and I was losing my motivation. This time I decided to do it the right way. I retought the concept of the program and planned carefully each step. I hope this is the last rewritting...

2.2 This version

2.2.1 Concept

As stated before, I retought the concept of m4x. In the beginning, it was just a macro processor for XML[1] and that guided the first versions. After thinking about the real purpose of the program and my needs, I came up with two possible situations. The extraction of information from XML files was a common goal in both scenarios. The only difference between them was the possibility of specifying the formatation of the data extracted.

Both situations had advantages and problems. If the program only extracted the information it would be simpler and faster, but would not solve the problem. Even with the possibility of specifying what information would be presented, we would have another parsing/interpreting phase to get the results we wanted.

It seemed that the best choice was to let the user specify the formatation of the data extracted from the XML file. The concept I came to was of *insertions*. Whenever a markup is found, the user specified information is inserted in the final document. There are two possible forms of listing this relations: relative or absolute.

The relative form is the direct implementation of the insertions concept. It would be a list of conditions (markup names, attribute names or values, ...) and the respective insertion.

The absolute form is somewhat subtle, but more intuitive. Instead of a list, there would be a template document and insertions placed in the positions where they will be inserted.

Although its implementation is a little more complex, the absolute form is the easiest way to describe the wanted final result. This is the form that will be used in this version of m4x.

Chapter 3

Design

3.1 The template document

The template document is a simple plain text file (Unicode support may be added in the future). The output generated by m4x actually is a copy of the template document, with the insertions expanded to their respective values.

All commands may, in general, appear anywhere in the template document and have the syntax:

`{ command` *`[arguments]`* `}`

Where *command* is one of the reserved words in table 3.1 and *arguments* is an optional list of arguments to the command.

markup	value	xml
content	file	expand
attribute	template	end

Table 3.1: Reserved words that may be used in an insertion.

The commands split into three main groups: XML insertions, that insert an element of a XML file; file insertions, that insert a file; and control commands.

3.1.1 XML insertions

`markup`

Syntax

`{ markup` *`[selection]`* `}`

Description

The `markup` reserved word inserts all markup names matching the markup description.

Example

The following example inserts all markup names that have a *src* attribute with a value in the *foo* domain.

```
{ markup ‘‘*’’ . ‘‘src’’ = ‘‘^[a-z]\+://foo.com’’ }
```

If put in the template document, it would insert the name of the first markup below, `img`, but would not insert the name of the second.

```
<img src=’’http://foo.com/images/title.png’’ />
<abc src=’’ftp://xyz.com/pub/README’’>
```

`content`

Syntax

```
{ content [selection] }
```

Description

`content` inserts all text inside (between the start element and the end element) markups that match the markup description. All child markups and their content are ignored and, thereby, not inserted in the final document.

Example

The example below inserts the content of all paragraph markups in an HTML file.

```
{ content ‘‘p’’ }
```

Below are some possible markups:

```
<p>Hello? World!</p>
<p align=’’right’’>One <b>Two</b> Three Four</p>
```

And this is what would be inserted:

```
Hello? World!
One  Three Four
```

`attribute`

Syntax

```
{ attribute [selection] }
```

Description

The `attribute` reserved word inserts all attribute names that match the selection.

Example

The following example inserts all attribute names found in the `body` markup of an HTML file.

```
{ attribute ``body'' }
```

The following markup:

```
<body bgcolor=''#ffffff'' hlink=''#00cc00''>
```

Would then "insert":

```
bgcolor
hlink
```

`value`

Syntax

```
{ value [selection] }
```

Description

The `value` reserved word inserts all attribute values that match the selection.

Example

The example below insert the `bgcolor` attribute value of the HTML example in the *attribute* reserved word definition:

```
{ value ``bgcolor'' }
```

This is what would be inserted:

```
#ffffff
```

3.1.2 File insertions

`file`

Syntax

```
{ file filename }
```

Description

The `file` reserved word inserts a file in the final document. Note that the inserted file is not interpreted in any way. It is just inserted "as is".

Example

The following example inserts the file named *abc.txt* in the final document.

```
{ file ``abc.txt'' }
```

`template`

Syntax

```
{ template filename }
```

Description

The `template` reserved word inserts a template file in the actual position of the template document. The inserted file is interpreted as it was directly typed in the template document.

Example

The following example inserts the file named *xyz.inc* in the template document.

```
{ template ‘‘xyz.inc’’ }
```

3.1.3 Control commands

`xml`

Syntax

```
{ xml filename }
```

Description

This reserved word adds the specified file name to the list of XML input files. After this point, all the information contained in the XML file may be inserted in the final document.

Example

This example adds the file named *foo.xml* to the list of XML input files.

```
{ xml ‘‘foo.xml’’ }
```

`expand`

`end`

Syntax

```
{ expand [here | inplace] [selection] }
{ end }
```

Description

The `expand` command interprets all text until its closing command `end` for each of the markups that match the selection. Inside the `expand` loop, the matched markup can be referred as `$` (dollar symbol) in the *selection* argument of the immediate inner commands.

The `expand` loop is restricted to the actual parent markup. Note that `expand` loops may be nested.

If the `here` modifier is used, the text is inserted in the actual location in the template document. If `inplace` is used, any text is inserted in the content of the parent markup of the referred markup. If no modifier is used, here is assumed (this behavior may be changed by the user).

Example

The example below copies all paragraphs of an HTML file and copies all unnumbered lists found.

```
First the paragraphs:

{ expand ``p'' }{ expand inplace ``b'' }*{ content $ }*{ end }{ content $ }
{ end }

Then the lists:

{ expand ``ul'' }{ expand ``li'' }{ content $ }
{ end }
{ end }
```

The following HTML file:

```
<html>
<head><title>Hi there</title></head>
<body>
<p>Hello? World!</p>
<ul>
  <li>Earth</li>
  <li>Venus</li>
  <li>Jupiter</li>
</ul>
<p>One <b>Two</b> Three <b>Four</b>. North East South West.</p>
<p>Bye...</p>
<ul>
  <li>Monday</li>
  <li>Tuesday</li>
  <li>Wednesday</li>
  <li>Thursday</li>
  <li>Friday</li>
  <li>Saturday</li>
  <li>Sunday</li>
</ul>
</body>
</html>
```

Would generate the following output:

```
First the paragraphs:

Hello? World!
One *Two* Three *Four*. North East South West.
Bye...

Then the lists:

Earth
Venus
Jupiter

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

3.1.4 Markup selections

Most of the commands available in the template document accept a *selection* as their argument. A selection is a way of describing a markup, its attributes and values. The syntax of a selection is as follows:

```
[ name ] [ . attribute ] [ = value ]
```

Where *name*, *attribute* and *value* are regular expressions, delimited by simple or double quotes. They represent the markup name, attribute names and attribute values, respectively. If anyone of these is omitted, the regular expression "`.`", which matches anything, is assumed.

If more than one markup is matched, the first one is used in the command. The following matches use the respective markups, in order of appearance. If `expand` is used, the text between the command and its terminator `end` is interpreted for each match.

All matched markups are inside the actual level of depth in the XML structure. This means that nested `expand` loops restrict the scope of the possible matching markups.

Lets look at some examples. All of them refer to the following XML extract:

```
<p>
Hi there!
  <q>One <r>book</r></q>
  <q>Two <r>dogs</r></q>
```

```
  <q>Three <r>cats</r></q>
</p>
<p>Empty</p>
```

The following block of commands are equivalent:

```
{ content ``p'' }
{ content ``p'' }

{ expand here  ``p'' }{ content $ }
{ end }

{ expand inplace ``p'' }{ content $ }
{ end }
```

And would produce the output below:

```
Hi there!
Empty
```

If one wanted the markups inside the *p* markups, he/she would use:

```
{ expand ``p'' }{ expand inplace ``.'' }{ content $ }{ end }{ content $ }
{ end }
```

That outputs:

```
Hi there!
One
Two
Three
Empty
```

Which is different from:

```
{ expand ``p'' }{ content $ }{ expand inplace ``.'' }{ content $ }{ end }
{ end }
```

That outputs the text below.

```
Hi there!
Empty
```

This is because the `expand` command with the `inplace` modifier only modifies the behavior of the commands after it.

3.2 The XML source files

All information inserted in the final document, as specified in the template document (see section 3.1), is extracted from XML source files[1]. The XML processor that m4x will use does not need to be a validating processor, but that would be a plus. Unicode support may be added as well, but is not a requirement.

3.3 The program interface

Chapter 4

Implementation

Chapter 5

Tests

Bibliography

[1] Extensible Markup Language (XML) 1.0 - Second Edition, by the World Wide Web Consortium. Latest version at `http://www.w3.org/TR/REC-xml`