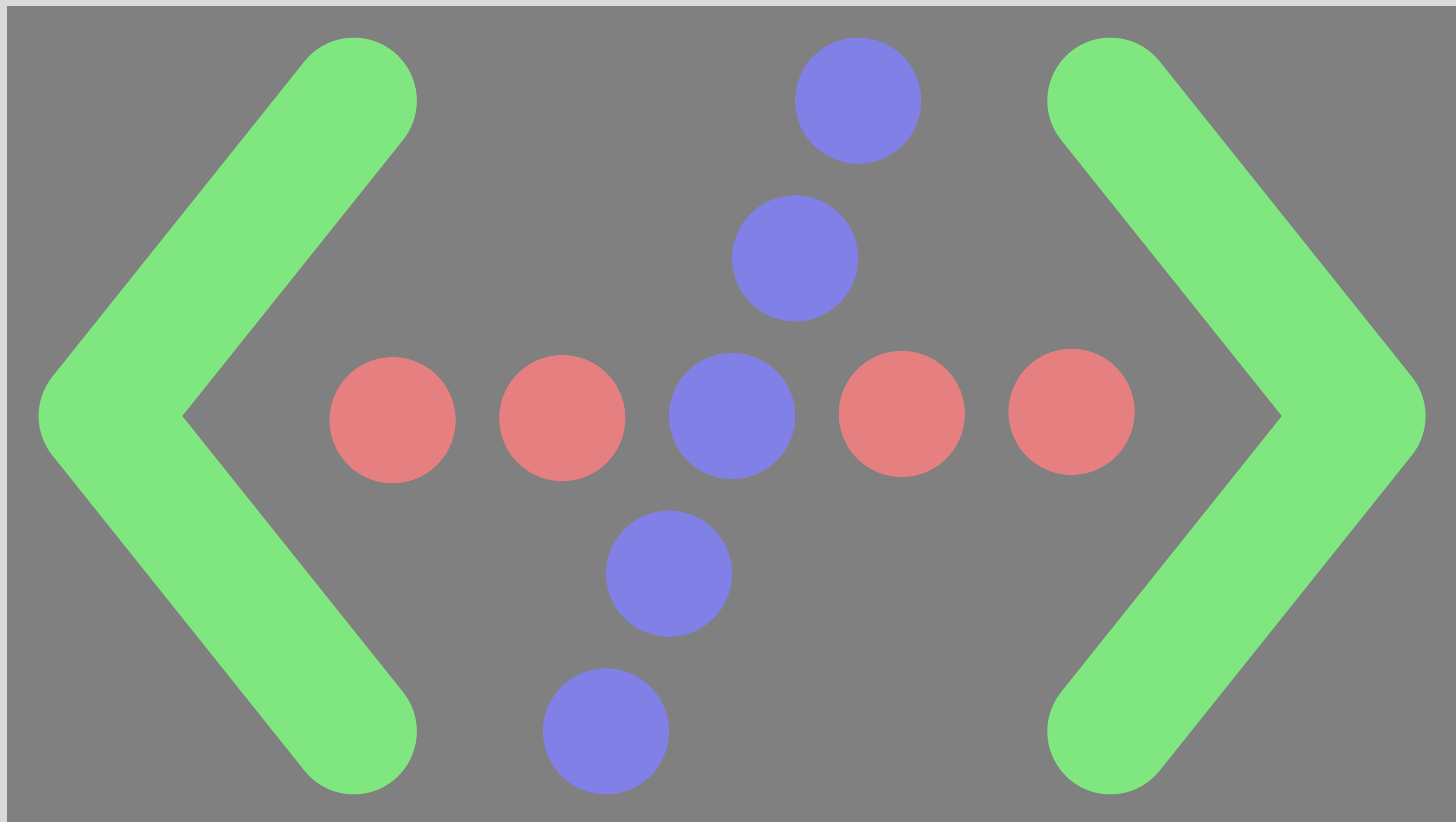
XML in ConTEXt

introduction
general markup
processing files
defining interfaces
basic workflows
some examples
command reference

PRAGMA ADE | November 9, 2001



introduction

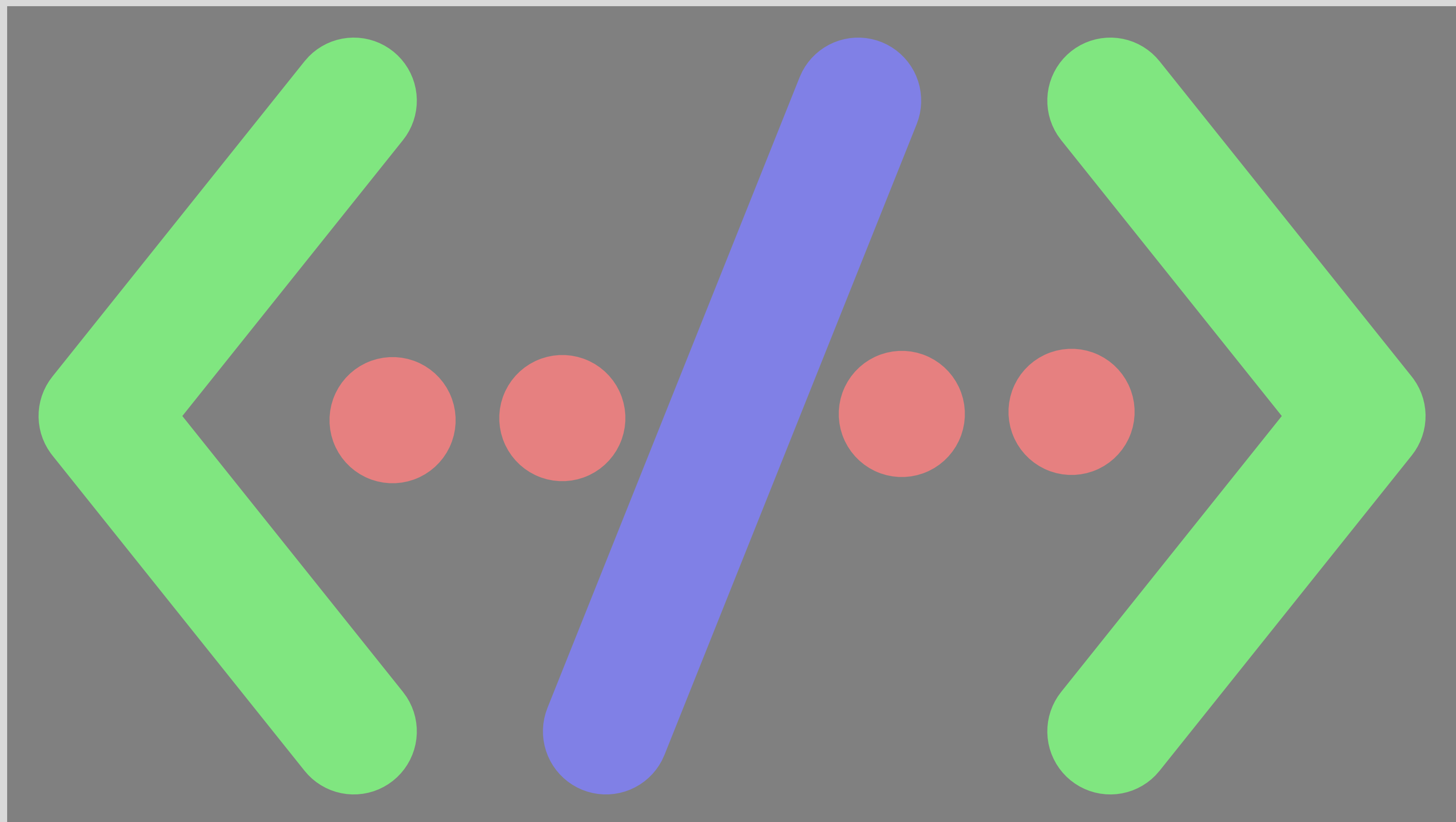
This document is meant for those who want to use ConTEXt for processing documents coded in XML. We will introduce some basics of XML processing first. In the following chapters we will explain the interface between XML and ConTEXt. This document is a by–product of the internal Pragma ADE project that is carried under the name eXaMpLe, so you may expect some examples.

Some basic knowledge of general markup as well as ConTEXt does not hurt. Since everyone who read this document probably spent some time using ConTEXt, we hope that the examples will ring a few bells.

We will *not* explain XML, XSLT, TEX, ConTEXt and PDF in full detail. Maybe the next chapters will inspire the reader to download the specifications and manuals and spend some time studying.

No, this document was not encoded in XML, but in TEX. The XML examples are processed as embedded XML, so what we keyed is what you see. We hope that you will enjoy reading this manual.

Hans Hagen & Ton Otten



general markup

General markup can be defined as “coding a document source in such a way that by reading the code, the structure and meaning is clear”. Readability of such a source is therefore an important prerequisite. In this chapter we will introduce a few concepts. A more in–depth exploration of mapping XML onto ConTEXt will take place in following chapters.

coding in angle brackets

An intentional way of coding is not new. Ever since computers were around, users have tried to code documents in such a way that its content could be understood by humans and programs. As soon as the coding permits multiple output formats, it can be classified as general markup. General markup has nothing to do with typography, but is focused on structure.

```
@chapter General Markup

In this chapter we will deal with ...
```

In this example, the `@chapter` ensures that we can pick up the chapter title and typeset it in a special way. To TEX, a more natural syntax is:

```
\chapter {General Markup}

In this chapter we will deal with ...
```

Both examples share that they not only enable parsing the code, but also provide a visual cue about the structure of the document. Although some people tend to classify this kind of coding as archaic, in principle it does not differ too much from todays most popular coding ‘languages’, like HTML and XML:

```
<chapter>General Markup</chapter>

In this chapter we will deal with ...
```

One of the purposes of coding in such a verbose way is that we want to be able to reuse the information. Since both the `@` and `\abc{}` way of coding is emotionally related to applications, some people favour the `<xyz>` way of dealing with source code. Although we feel that many applications that deal with this angle bracketed code, are just as ‘narrow minded’ as any application, we nevertheless follow this thread in this document.

Just adding <> to a document is not enough. A document may look modern and up-to-date, but this kind of neutral coding only makes sense when one is willing to look behind the horizon of todays applications. Typesetting and reading are moving from paper to screen and beyond, and the more structure we add, the more prepared we are for future reuse. But, how much structure is needed? The previous example can be recoded as:

```
<chapter>General Markup</chapter>
<p>In this chapter we will deal with ...</p>
```

Even these additional paragraph markers cannot be enough, so maybe we should go further:

```
<chapter>
  <title>General Markup</title>
  <text>
    <p>In this chapter we will deal with ...</p>
  </text>
</chapter>
```

Coding in this way, makes it easy to filter chapters, as well as their titles, the text or even individual paragraphs. It may be clear that in order to handle such code properly, we also need a clear description on how to interpret the labels. Hopefully, the labels themselves provide some information, but it is good practice to define them in a so called document type definition:

```
<?XML version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT chapter (title,text)>
<!ELEMENT text    (p)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT p       (#PCDATA)>
```

This means as much as: there can be `chapter`s, and such a chapter can have a `title` and a body of `text`. A text can consist of one or more paragraphs, and since there can be many of them, we use a short element name `p`.

converting XML

In itself, coding and defining is not enough: we want to typeset a document, and why not use TEX for doing that? Later we will describe several ways of going from XML to for instance PDF by using TEX, but first we will concentrate on a general transformation language, in this case XSLT. The next snippets define the transformation from the previous code to ConTEXt code.

```
<xsl:template match="chapter">
<xsl:apply-templates/>
</xsl:template>
```

This is the way to say that everything between `<chapter>` and `</chapter>` is passed along the way.

```
<xsl:template match="chapter/title">
<xsl:text>\chapter{</xsl:text>
<xsl:apply-templates/>
<xsl:text>}</xsl:text>
</xsl:template>
```

The title is to be converted to `\chapter{...}`, and the text is simply passed on.

```
<xsl:template match="chapter/text">
<xsl:apply-templates/>
</xsl:template>
```

To make sure that TEX will know where a new paragraph will start, the `</p>` tags are converted into `\par`'s.

```
<xsl:template match="chapter/text/p">
<xsl:apply-templates/>
<xsl:text>\par</xsl:text>
</xsl:template>
```

A few more commands are needed to make this snippet of XSLT acceptable for programs like xt, but when processed, we will have a proper TEX file that can be processed by CONTEXT.

direct processing

In many cases, the overhead involved in additional coding is not worth the effort. Since TEX and CONTEXT can manipulate text in quite advanced ways, directly coding in a syntax more natural to TEX, often pays off. When properly coded, such a document can be converted to XML anyway.

In CONTEXT you can also mix TEX and XML code. You can for instance decide to encode your formulas in MATHML, while the rest is just TEX. From this you may indeed deduce that CONTEXT can handle XML directly. Therefore, in many cases, you can bypass the XSL conversion stage.

```
<math>
  <apply> <root/>
    <cn> 3 </cn>
    <apply> <times/>
      <apply> <plus/>  <ci> a </ci> <ci> b </ci> </apply>
      <apply> <minus/> <ci> c </ci> <ci> d </ci> </apply>
    </apply>
  </apply>
</math>
```

CONTEXT will convert this piece of content MATHML into:

$$\sqrt{3\,(a+b)\,(c-d)}$$

Although XML parsing is part of the core, you explicitly need to load the specific kind of XML support you need for your job. You define the mapping of XML elements onto CONTEXT commands in your regular environments. However, for some well standardized DTD's CONTEXT (will) provide the corresponding XML filters. Basic MATHML support is for instance part of the CONTEXT distribution, although we can imagine that you use your own implementation. The special file prefix xtag is reserved for the filters provided by PRAGMA ADE. You can load the presentational and content MATHML filters by saying:

```
\useXMLfilter[mml,mmp,mmc] % basic mathml/presentational/content
```

getting started

A convenient way to get familiar with XML is to embed this kind of coding into a normal TEX document. In this manual, we used buffers to accomplish this. So, given that you have loaded the MATHML filters, you can say:

```
\startbuffer
<math>
  <apply> <eq/>
    <apply> <power/> <ci> a </ci> <ci> 2 </ci> </apply>
    <apply> <plus/>
      <apply> <power/> <ci> b </ci> <ci> 2 </ci> </apply>
      <apply> <power/> <ci> c </ci> <ci> 2 </ci> </apply>
    </apply>
  </apply>
</math>
\stopbuffer
```

You can show this buffer verbatim by saying \typebuffer and process its content with \processXMLbuffer. As with other buffers, you can use label them and recall them by name.

$$a^2 = b^2 + c^2$$

Instead of buffers, you can also use:

```
\startXMLdata
<math>
  <apply> <plus/> <ci> a </ci> <ci> b </ci> <ci> c </ci> </apply>
</math>
\stopXMLdata
```

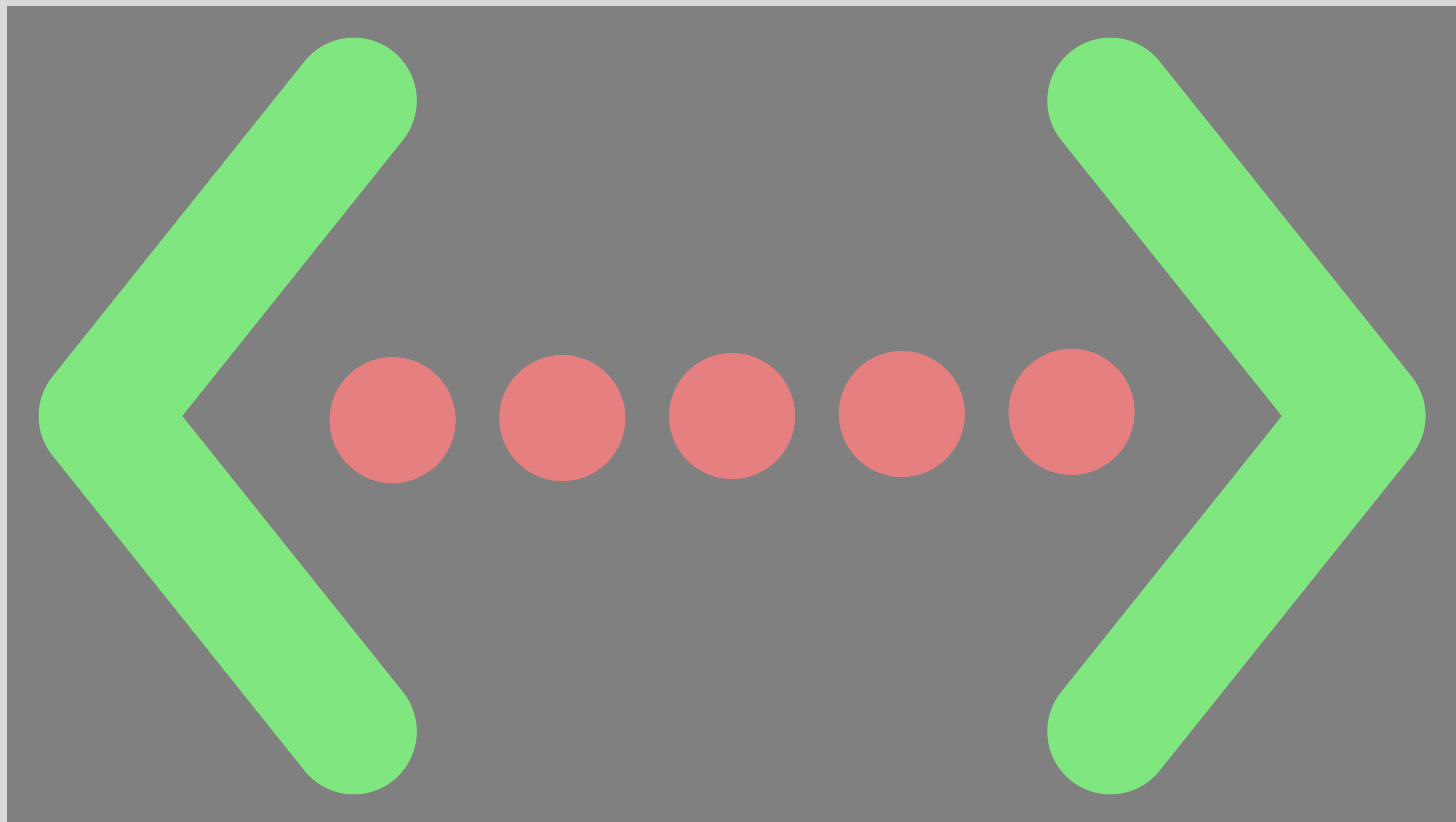
In print this wil become:

$$a + b + c$$

You can save XML code in memory with:

```
\startXMLcode[somename]
<math>
  <apply> <plus/> <ci> a </ci> <ci> b </ci> <ci> c </ci> </apply>
</math>
\stopXMLcode
```

after which you can recall it with `\getXMLcode [somename]`, for instance in the body of a macro. This latter mechanism is meant for future communication with other applications.



processing files

Apart from the previously mentioned methods of embedding XML into a document, we need a way to process a complete XML file. In doing that, we have the problem that once XML processing mode is entered, we no longer can use the backslash to execute commands. So, we have to make sure that we not only can enter XML mode, but also quit it.

```
\defineXMLenvironment [note] {(} {)}
\enableXML just a <note>little</note> bit of XML
```

When we process this file, TEX will finally wait for you to enter a command on the console, since it encountered the end of a file. One solution is the following:

```
\defineXMLenvironment [note] {(} {)}
\enableXML just a <note>little</note> bit of XML
<?context-command \disableXML ?>
```

The next solution works as well:

```
\defineXMLenvironment [note] {(} {)}
\starttext \enableXML
just a <note>little</note> bit of XML
<?context-command \stoptext ?>
```

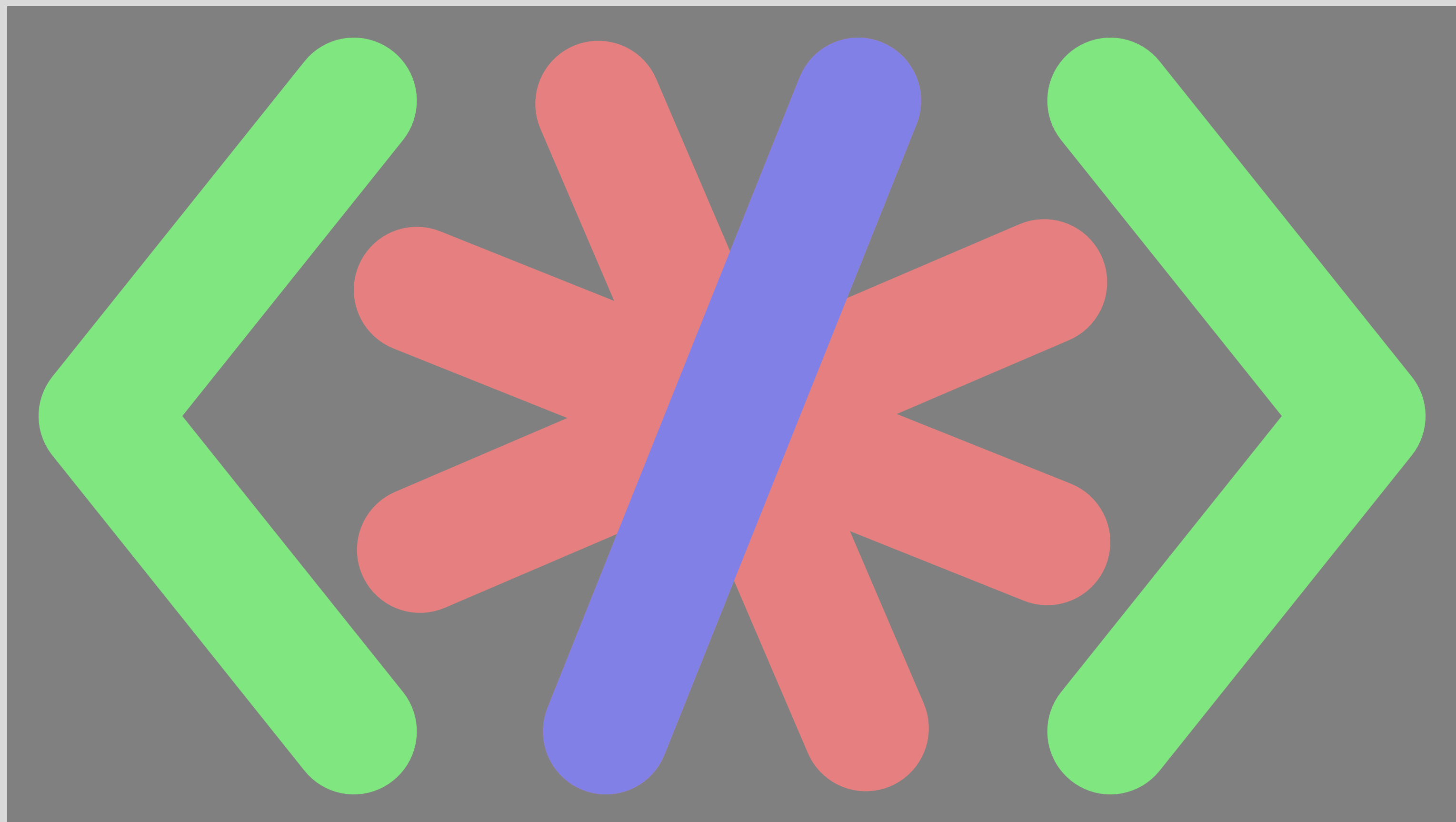
These methods are quite ugly, and since normally we will use XML source code that is stored in another file, in most cases one will use one of:

```
\processXMLfile        {filename.xml}
\processXMLfilegrouped {filename.xml}
```

This second command is probably the best one to use, as in:

```
% here go style definitions
% and XML to TEX mappings
```

```
\starttext
\processXMLfilegrouped {filename.xml}
\stoptext
```



defining interfaces

When mapping XML onto ConTEXt commands, the complexity of the definitions depends on to what extend the content is to be manipulated. We will start with some examples of straightforward mappings, where no tricky things are needed.

mapping elements

Any document has a starting point and an end point. Furthermore there will be many rather symmetrical structuring commands.

Consider the following example:

```
<document>
  <title name="small">A very small document</title>
  <list packed="yes">
    <item>of course these commands <b>don't</b> match &context;
          commmand names</item>
    <item>and even worse, <ref name="att"> attributes will not
          be the same as in &context;</item>
  </list>
</document>
```

For the moment we will ignore the attributes and just handle the plain elements. The document is encapsulated in a `document` element. You can use the begin- and endtag to do initializations, but for the moment we stick to a simple definition.

```
\defineXMLenvironment [document] \starttext    \stoptext
```

The title element comes next. Because we need to pass the content of this element to a context heading command, like `\title`, we will define it as an argument element.

```
\defineXMLargument [title] \title
```

This means as much as that the content is collected and passed on to \title. As usual, ConTEXt will take care of preparing the necessary table of content entries, if needed.

The itemized list has child elements that identify the individual components. A suitable definition is:

```
\defineXMLenvironment [list] \startitemize \stopitemize
\defineXMLenvironment [item] \item         \par
```

Because in ConTEXt you don't have to explicitly mark the end of an item, the last element can be defined as a command element:

```
\defineXMLcommand [item] \item
```

The b element can be mapped onto ConTEXt's \bf command. The best way to do this is the following:

```
\defineXMLgrouped [b] \bf
```

In this case we could have said:

```
\defineXMLenvironment [b] {\bgroup\bf} {\egroup}
```

But in other situations the previous definition is more robust. Although a valid definition, the following one is to be avoided, since picking up arguments takes time.

```
\defineXMLargument [b] {\groupedcommand{\bf}{}}
```

The reference element ref is an example of an empty element. The following invocations are equivalent:

```
<ref name="something"></ref>
<ref name="something" />
```

For handling empty elements, we have two definition commands available. Given that \myXMLref is defined, you can say:

```
\defineXMLcommand  [ref] \myXMLref
\defineXMLsingular [ref] \myXMLref
```

The command definition handles both cases, while the singular one only handles the second (../>) alternative. This makes it possible to distinguish between both cases, although officially there is no distinction.

Now, how should we define this \myXMLref command. It is clear that we have to use the name attribute. Because the definition of \myXMLref is rather simple, we can instead directly handle the reference, as in:

```
\defineXMLsingular [ref] {\pagereference[\XMLpar{ref}{name}{}]}
```

You can pick up a parameter with \XMLpar. Each parameter is accessed by its element and attribute. The third argument to \XMLpar is the default value. We can slightly change the definition of the title element.

```
\defineXMLargument [title] {\title[\XMLpar{title}{name}{}]}
```

For more advanced attribute handling, there are specific testing macros available, and to enlighten the task of using the attributes as input for ConTEXt setup commands, there are methods to map values onto other ones.

A complication in mapping is that we have to make sure that the values are passed in the way that macros expect them. For instance, if we want to convert packed="yes" into the packed directive, we can use:

```
\XMLifequalelse{itemize}{packed}{yes}{packed}{}
```

This expands into packed or into an empty string. Expansion can be very confusing since one never knows when this takes place, unless one has a rather detailed knowlegde of ConTEXt's low level behaviour.

```
\defineXMLenvironment [itemize]
  {\startitemize[\XMLifequalelse{itemize}{packed}{yes}{packed}{}]}
  {\stopitemize}
```

The following definition works out all right, since \startitemize expands its argument, but in other situations one may have to force expansion:

```
\defineXMLenvironment [itemize]
  {\expanded{\startitemize[....]}}
  {\stopitemize}
```

This example also demonstrates that mapping XML onto ConTeXt is not that hard, but sometimes needs a bit of hard work. The good news is of course that once that job is done, you seldom have to look into it again.

entities

You can compare entities to ConTeXt's named glyphs, abbreviations, logos and alike. They are, so to say, predefined representations of something. In the previous example we met `&context;` as an example of an entity.

You can define an entity with:

```
\defineXMLentity [context] {The ConTeXt Macro Package}
```

although, when you use TeX, it makes more sense to say:

```
\defineXMLentity [context] \ConTeXt
```

We will provide an entity dictionary for the more or less standardized named glyphs, the most common logos, and special characters. You can load these by saying:

```
\useXMLfilter[ent]
```

As with elements, you can always overload an entity. Behind the screens, entities become empty elements, which enables them to move around in the ConTeXt kernel and auxiliary files without much trouble. So, don't be surprised when you see such a disguised entity passing by on your screen.

delimited environments

In TeX there are several ways to define a command that acts upon an argument. The most straightforward way is:

```
\def\MyCommand#1{... #1 ...}
```

Such a command is invoked with `\MyCommand{Works}` where the `{}` determines the begin and end of the argument.

Sometimes, using braces is not that charming, and delimiters are used. Many setup commands in ConTeXt use `[]` to delimit the parameters. One reason for this is that it stands out in editors, especially when they support syntax highlighting. Another reason is that implementing optional arguments is more convenient with non–braces.

Yet another kind of delimiting is the following:

```
\StartOfMyText
... my beautiful text ...
\StopOfMyText
```

Such a command can be defined as:

```
\def\StartOfMyText#1\StopOfMyText{... #1 ...}
```

Normally users of ConTeXt can remain happily unaware of how a command is implemented. There are however a few cases where ConTeXt uses delimited commands of the last category, for instance in natural tables. Since these macros were developed with processing HTML in mind, using them in an XML environment makes sense.

```
<table>
  <tr> <td> a </td> <td> one   </td> <td> alpha </td> </tr>
  <tr> <td> b </td> <td> two   </td> <td> beta  </td> </tr>
  <tr> <td> c </td> <td> three </td> <td> gamma </td> </tr>
</table>
```

Here we need to map `<tr> ... </tr>` onto `\bTR ... \eTR` and `<td> ... </td>` onto `\bTD ... \eTD`. This is handled by the pickup element environment.

```
\defineXMLenvironment [table] \bTABLE \eTABLE
\defineXMLpickup      [tr]    \bTR    \eTR
\defineXMLpickup      [td]    \bTD    \eTD
```

For simple tables this solution works quite well, which is demonstrated here.

a	one	alpha
b	two	beta
c	three	gamma

But, what happens when a user nests a table. Just take a close look at the following table, and try to imagine what happens when we pick up the argument of the <td> that is followed by <table>.

```
<table>
<tr> <td> a </td> <td> 1 </td> </tr>
<tr> <td> <table> <tr> <td> a </td> <td> 1 </td>
                       <td> b </td> <td> 2 </td>
                       <td> c </td> <td> 3 </td> </tr>
   </table> </td> <td> 2 </td> </tr>
<tr> <td> c </td> <td> 3 </td> </tr>
</table>
```

The argument will not contain the whole table, but just <table> <tr> <td> a since TEX will stop grabbing the argument when it encounters the </td>. Even if TEX would have a regular expression engine like PERL, handling this kind of nesting would be quite complicated.

In a document coded in TEX, putting {} around the inner table would solve the problem, so this is what we will do behind the screen.

```
\defineXMLenvironment [table] \bTABLE \eTABLE
\defineXMLnested      [tr]    \bTR    \eTR
\defineXMLnested      [td]    \bTD    \eTD
```

The nested environment analyzes its argument and keeps picking up code until the nesting is proper. Because this slows down processing, you should not use the nested environment unneeded.

a	1
a 1 b 2 c 3	2
c	3

This also works with more complicated cases, like the following (unreadable) table:

```
<table>
<tr> <td> a </td> <td> 1 </td> </tr>
<tr> <td> <table> <tr> <td> a </td> <td> 1 </td>
                       <td> <table>
                              <tr> <td> a </td> <td> 1 </td>
                                   <td> b </td> <td> 2 </td>
                                   <td> c </td> <td> 3 </td> </tr>
                     </table> </td> <td> 2 </td>
                       <td> c </td> <td> 3 </td> </tr>
   </table> </td> <td> 2 </td> </tr>
<tr> <td> c </td> <td> 3 </td> </tr>
</table>
```

a	1
a 1 a 1 b 2 c 3 2 c 3	2
c	3

pushing and popping

```
<text>
```

```
<title>Hello <tt>\World</tt>, this is &tex;!</title>

Welcome to this small XML example.

<figure label="cow">
  <caption>Some Caption</caption>
  <graphic>koe.pdf</graphic>
</figure>

<!-- Next comes some rather contextual code. The ? indicates
that we are dealing with a processing instruction. -->

<?context-command {\bf What a happy end to a small story!} ?>

</text>
```

We will start with the easy part of the interface to this XML document.

```
\defineXMLenvironment [text]  {\starttext} {\stoptext}
\defineXMLargument    [title] {\chapter[\XMLpar{title}{label}{}]}
\defineXMLgrouped     [tt]    {\tt}
```

In order to resolve some entities, we also say:

```
\useXMLfilter[ent]
```

The comment is skipped, while the processing instruction is processed right away. So, how about the graphic.

```
\defineXMLenvironment [figure]
  {\bgroup
   \defineXMLpush[caption]
   \defineXMLpush[graphic]}
  {\placefigure
     [\XMLpar{figure}{location}{here}]
     [fig:\XMLpar{figure}{label}{unknown}]
     {\XMLpop{caption}}
     {\externalfigure[\XMLpop{graphic}]}
   \egroup}
```

Before we can process the graphic, we need to know its caption. Therefore, we push the caption and the graphic definition into memory and at the end pop them in the order needed.

more on attributes

In the previous example, the location is passed as it is. If the XML code is not yours, there is a good chance that the values don't match those understood bij ConTeXt. This omission can be compensated by mapping a value, as in:

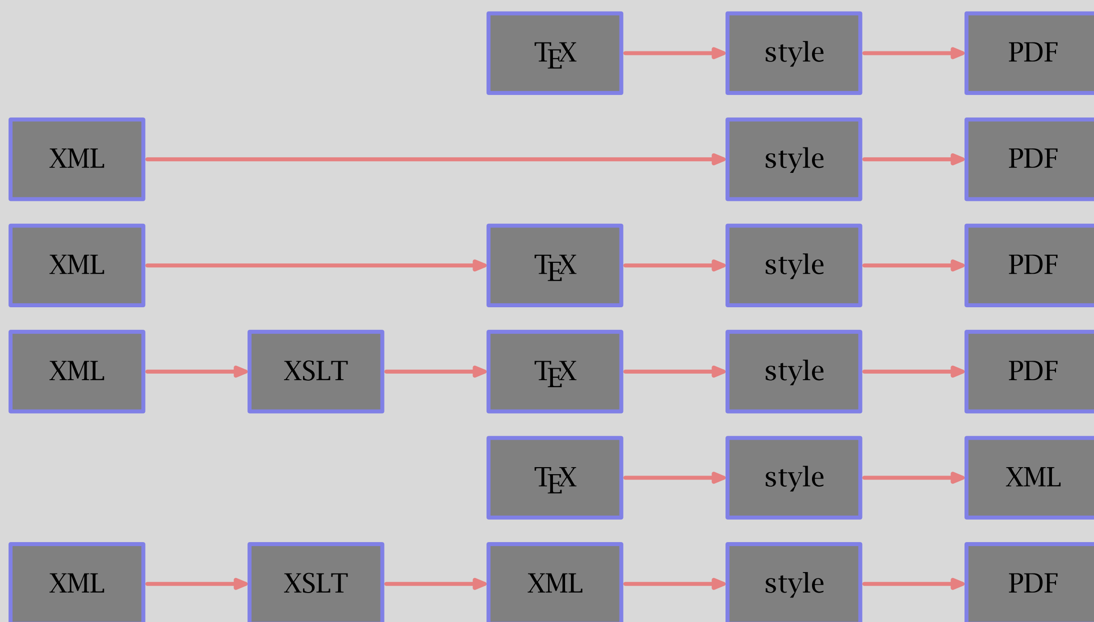
```
\mapXMLvalue{figure}{location}{thisplace}{here}
```

When we replace the `\XMLpar` with `\XMLvar`, we get the remapped value. This would permit calls like:

```
<figure label="cow" location="thisplace"> ... </figure>
```

basic workflows

We won't be surprised if, at this stage, you have lost track on how to proceed in going from XML to TEX. This chapter will describe several methods, and which way you go depends on both your input and your wishes.



This manual was processed this way. The source was coded in TEX, but in order to demonstrate how XML is processed, we used snippets of XML. In such cases, the necessary XML support is part of the regular style file.

```
texexec --pdf yourfile
```

The most straightforward way to process XML is to let ConTeXt directly interpret the XML commands. Normally you will use a specific environment file in which the mapping as well as the style setup is defined.

```
texexec --pdf --env=list yourfile.xml
```

In this case, TeXexec will make a dummy file called `yourfile.tex` that loads the XML file. If you simply want to test a MathML fragment, you can say:

```
texexec --pdf --xmlfilter=mml,mmp,mmc yourfile.xml
```

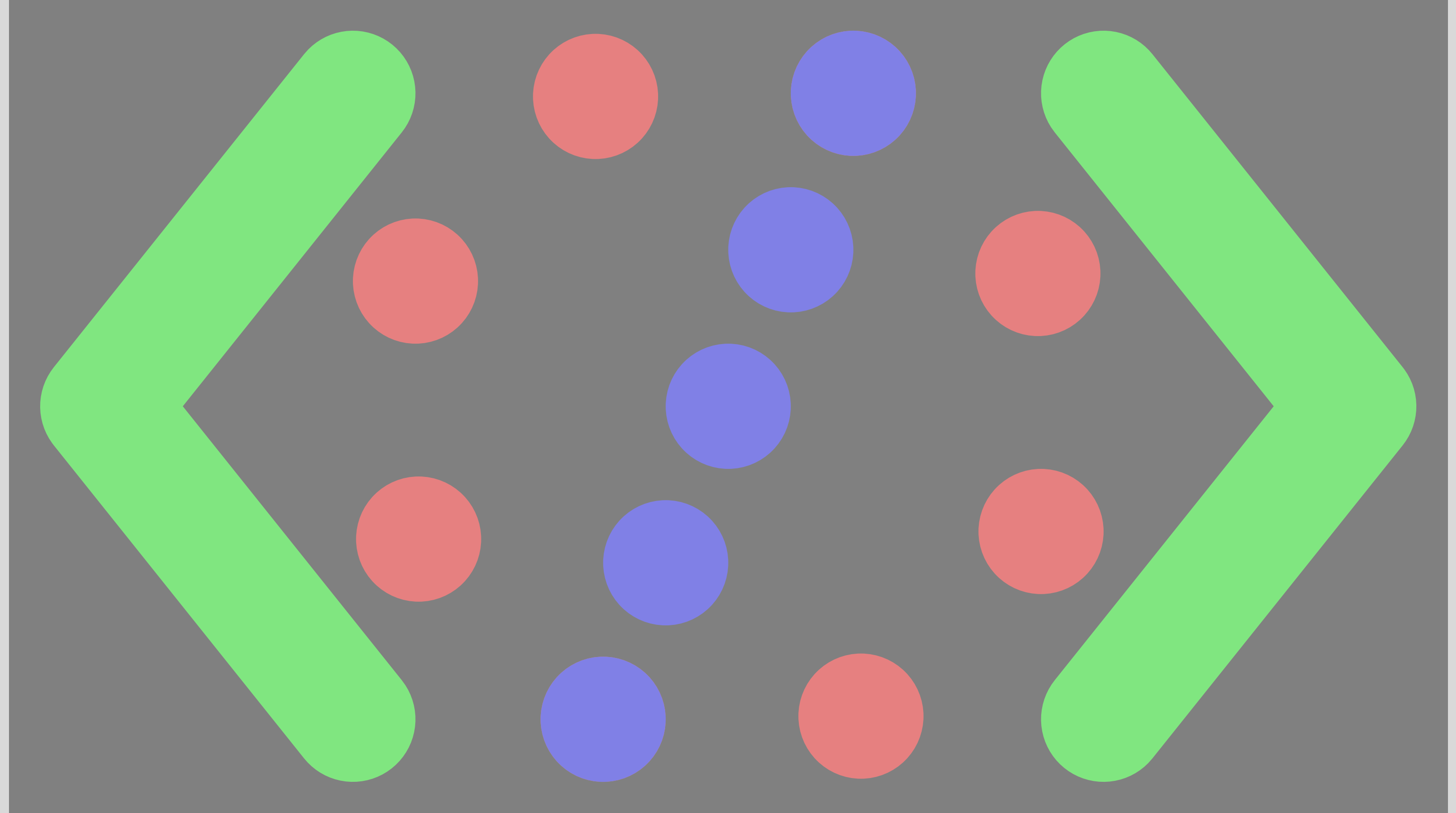
This method uses a dedicated converter that converts the XML code into a TEX file where angle brackets and ampersands are replaced by a few TEX commands and where characters that have a special meaning in TEX become entities. In most cases, the direct interpretation of XML code is more suitable, but for less restrictive dialects this is a safe way to go. Discussing this method, which is occasionally used at PRAGMA ADE is beyond the scope of this document.

This can be a convenient method but its usage depends on the complexity of the mapping. Although using XSLT has a certain charm, a decent mapping file can look quite unreadable and take some time to prepare. This method is used when manipulations have to take place that are not present in ConTEXt. The resulting TEX file is processed as any normal TEX file.

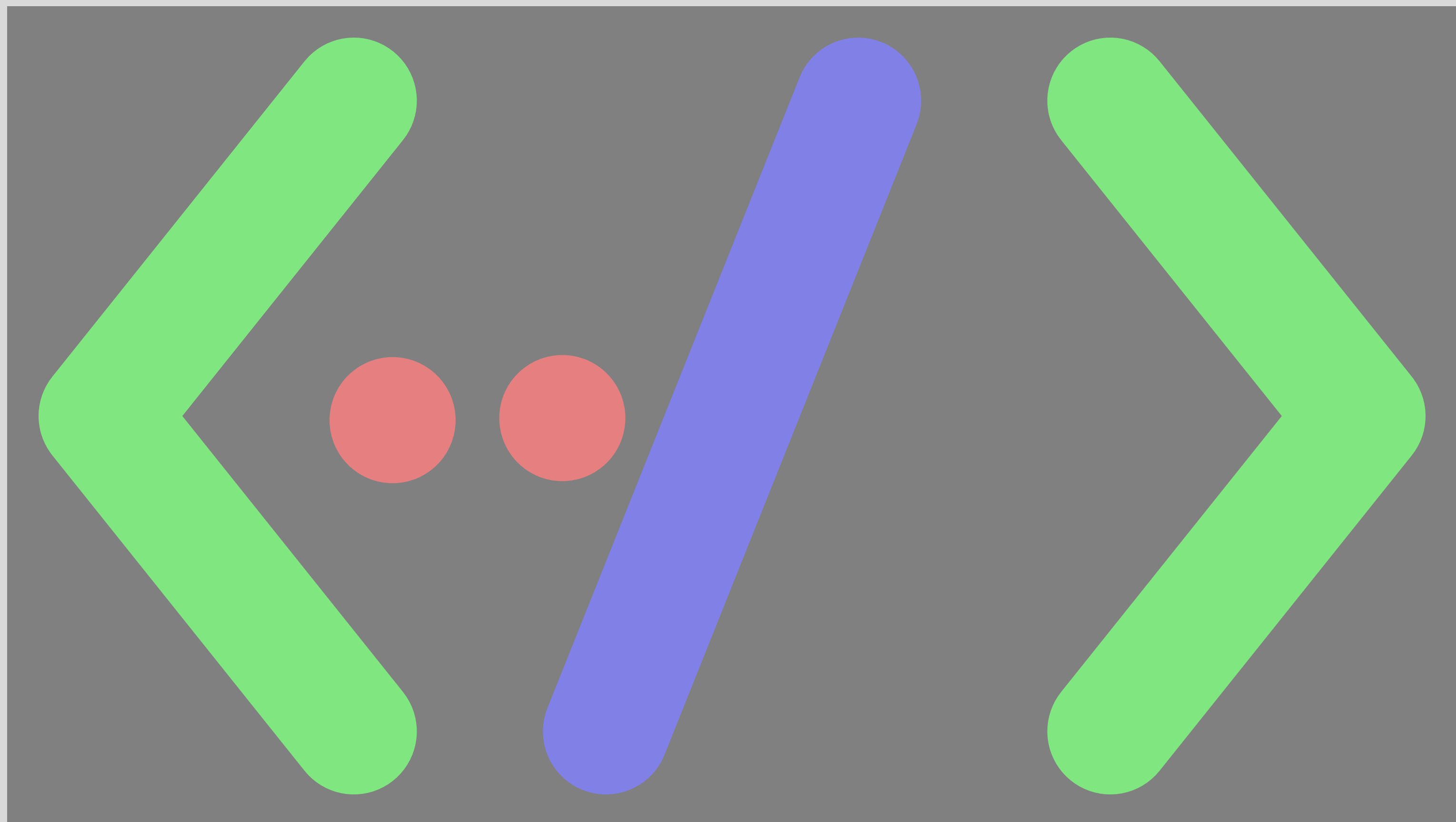
When you want to do a lot of manipulations, or when you want to convert part of the XML code into other code (think of manipulating math), you can use XSLT to do that before the document source is passed on to ConTeXt. It is to be expected that such filters will become available.

When you want to convert TEX documents into XML (or HTML) counterparts, it makes sense to use TEX itself, because that way we have the full engine at our fingertips.

In ConTEXt we use a strange but charming way to handle this conversion: we simply typeset XML. The typeset document (with lots of angle brackets) is then converted into raw text. A complication is math, but given that you can embed MathML in your document, this can be passed on directly.



examples of markup



command reference

In this chapter we summarize the mapping commands. You should be aware of the fact that some of them are experimental and so their names and even their meanings may change.

For the moment we left out the commands that are used for two-step remapping MATHML onto TEX. Once these have proven to be useful, they will be discussed here too. These two-step remapping commands convert an XML stream into a TEX one that then can be processed in a for TEX more comfortable way.

mapping elements

```
\defineXMLsingular    [name] {command}
\defineXMLcommand     [name] {command}
\defineXMLgrouped     [name] {command}
\defineXMLargument    [name] {command}
\defineXMLargument    [name] {command}
\defineXMLpickup      [name] {before} {after}
\defineXMLenvironment [name] {before} {after}
\defineXMLpush        [name]
\defineXMLprocess     [name]
\defineXMLnested      [name] {before} {after}
```

instruction processors

```
\defineXMLprocessor [name] \command
```

The following processing instructions are predefined:

```
<?context-command   \somecommand{to be executed} ?>
<?context-directive class key value              ?>
<?context-message   some text to show in the log ?>
```

mapping entities

```
\defineXMLentity [name] {command}
```

processing files

```
\XMLinput               filename
\processXMLfile        {filename}
\processXMLfilegrouped {filename}
\showXMLfile           {filename}
```

embedding code

```
\startXMLdata xml commands \stopXMLdata

\processXMLbuffer [name]

\startXMLcode [name] xml commands \stopXMLcode
\getXMLcode   [name]
```

loading filters

```
\useXMLfilter [list]
```

handling attributes

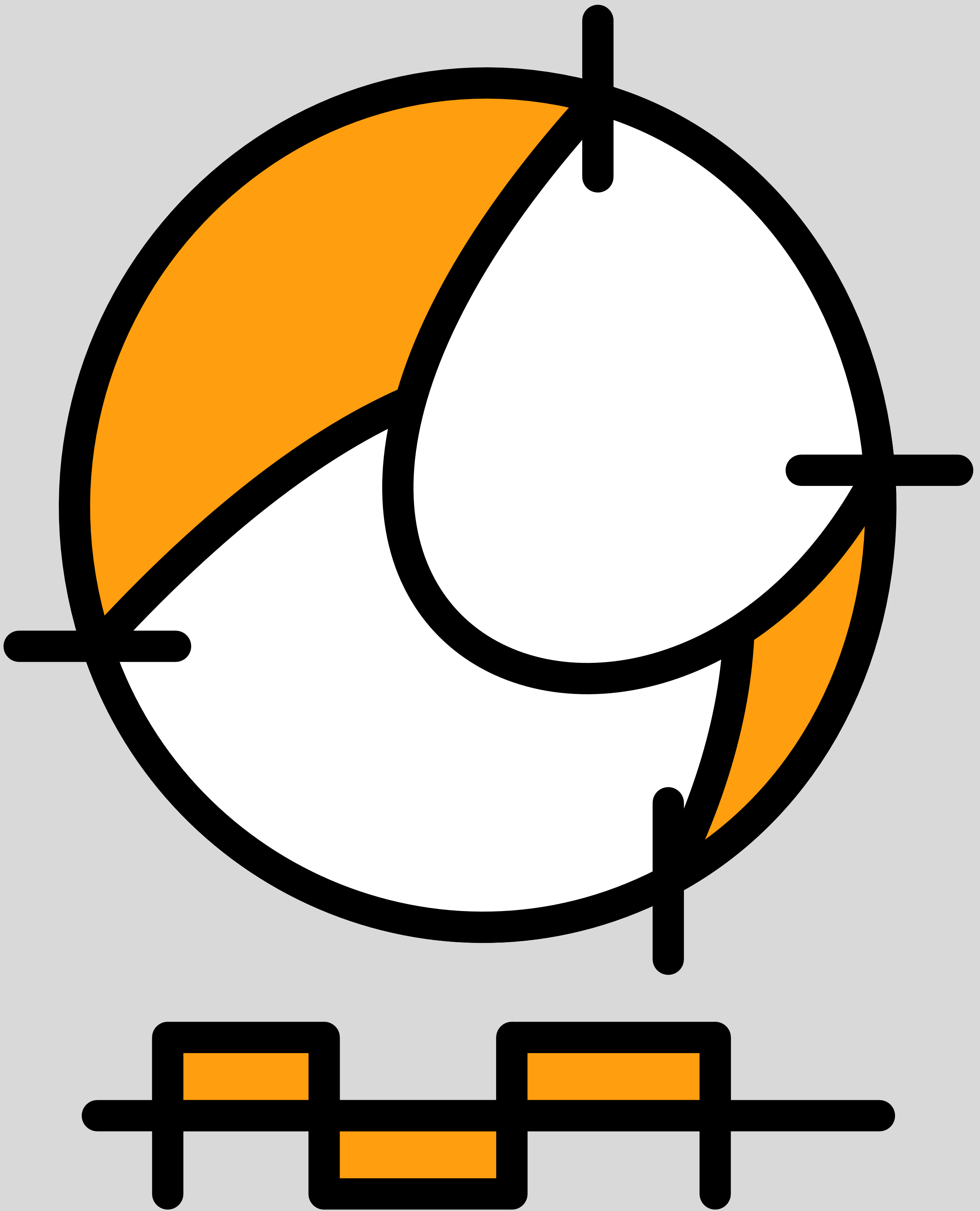
```
\mapXMLvalue {name} {key} {old} {new}

\XMLvar {name} {key} {default}
\XMLval {name} {key} {default}
\XMLpar {name} {key} {default}

\doifXMLvarelse {name} {key} {then branch} {else branch}
\doifXMLvalelse {name} {key} {then branch} {else branch}
\doifXMLparelse {name} {key} {then branch} {else branch}

\getXMLarguments {class} {key="value" pairs}

\XMLifequalelse {name} {key} {value} {then branch} {else branch}
```

manipulating pushed elements

```
\XMLpop     {name}
\XMLappend  {name} {text}
\XMLprepend {name} {text}
\XMLerase   {name}
\XMLassign  {name}
\XMLshow    {name}

\defXMLstring \command {name}

\doifXMLdataelse {name} {then branch} {else branch}
```

auxiliary commands

```
\defXMLlowerclean \command

\bXMLs
\eXMLs
```



PRAGMA Advanced Document Engineering

www.pragma-ade.com