Dealing with XML in ConTeXt MkIV

Hans Hagen, Pragma ADE, August 14, 2011

Contents

Introduction

This manual presents the MkIV way of dealing with XML. Although the traditional MkII streaming parser has a charming simplicity in its control, for complex documents the tree based MkIV method is more convenient. It is for this reason that the old method has been removed from MkIV. If you are familiar with XML processing in MkII, then you will have noticed that the MkII commands have `XML` in their name. The MkIV commands have a lowercase `xml` in their names. That way there is no danger for confusion or a mixup.

You may wonder why we do these manipulations in TEX and not use XSLT (or other transformation methods) instead. The advantage of an integrated approach is that it simplifies usage. Think of not only processing the document, but also using XML for managing resources in the same run. An XSLT approach is just as verbose (after all, you still need to produce TEX code) and probably less readable. In the case of MkIV the integrated approach is also faster and gives us the option to manipulate content at runtime using LUA. It has the additional advantage that to some extend we can handle a mix of TEX and XML because we know when we're doing one or the other.

This manual is dedicated to Taco Hoekwater, one of the first CONTEXT users, and also the first to use it for processing XML. Who could have thought at that time that we would have a more convenient way of dealing with those angle brackets. The second version for this manual is dedicated to Thomas Schmitz, a power user who occasionally became victim of the evolving mechanisms.

Hans Hagen
PRAGMA ADE
Hasselt NL
2008–2016

< 1 > Setting up a converter

<< 1.1 >> from structure to setup

We use a very simple document structure for demonstrating how a converter is defined. In practice a mapping will be more complex, especially when we have a style with complex chapter openings using data coming from all kind of places, different styling of sections with the same name, selectively (out of order) flushed content, special formatting, etc.

```
<?xml version='1.0' standalone='yes?>

<document>
  <section>
    <title>Some title</title>
    <content>
      <p>a paragraph of text</p>
      <p>another paragraph of text</p>
    </content>
  </section>
</document>
```

Say that this document is stored in the file `demo.xml`, then the following code can be used as starting point:

```
\startxmlsetups xml:demo:base
  \xmlsetsetup{#1}{*}{-}
  \xmlsetsetup{#1}{document|section|p}{xml:demo:*}
\stopxmlsetups

\xmlregisterdocumentsetup{demo}{xml:demo:base}

\startxmlsetups xml:demo:document
  \starttitle[title={Contents}]
    \placelist[chapter]
  \stoptitle
  \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:demo:section
  \startchapter[title=\xmlfirst{#1}{/title}]
    \xmlfirst{#1}{/content}
  \stopchapter
\stopxmlsetups

\startxmlsetups xml:demo:p
```

```
  \xmlflush{#1}\endgraf
\stopxmlsetups

\xmlprocessfile{demo}{demo.xml}{}
```

Watch out! These are not just setups, but specific XML setups which get an argument passed (the `#1`). If for some reason your XML processing fails, it might be that you mistakenly have used a normal setup definition. The argument `#1` represents the current node (element) and is a unique identifier. For instance a `<p>..</p>` can have an identifier demo::5. So, we can get something:

```
\xmlflush{demo::5}\endgraf
```

but as well:

```
\xmlflush{demo::6}\endgraf
```

Keep in mind that the references tor the actual nodes (elements) are abstractions, you never see those `<id>::<number>`'s, because we will use either the abstract `#1` (any node) or an explicit reference like `demo`. The previous setup when issued will be like:

```
\startchapter[title=\xmlfirst{demo::3}{/title}]
  \xmlfirst{demo::4}{/content}
\stopchapter
```

Here the `title` is used to typeset the chapter title but also for an entry in the table of contents. At the moment the title is typeset the XML node gets looked up and expanded in real text. However, for the list it gets stored for later use. One can argue that this is not needed for XML, because one can just filter all the titles and use page references, but then one also looses the control one normally has over such titles. For instance it can be that some titles are rendered differently and for that we need to keep track of usage. Doing that with transformations or filtering is often more complex than leaving that to TEX. As soon as the list gets typeset, the reference (`demo::#3`) is used for the lookup. This is because by default the title is stored as given. So, as long as we make sure the XML source is loaded before the table of contents is typeset we're ok. Later we will look into this in more detail, for now it's enough to know that in most cases the abstract `#1` reference will work out ok.

Contrary to the style definitions this interface looks rather low level (with no optional arguments) and the main reason for this is that we want processing to be fast. So, the basic framework is:

```
\startxmlsetups xml:demo:base
  % associate setups with elements
\stopxmlsetups

\xmlregisterdocumentsetup{demo}{xml:demo:base}

% define setups for matches

\xmlprocessfile{demo}{demo.xml}{}
```

In this example we mostly just flush the content of an element and in the case of a section we flush explicit child elements. The `#1` in the example code represents the current element. The line:

```
\xmlsetsetup{demo}{*}{-}
```

sets the default for each element to 'just ignore it'. A `+` would make the default to always flush the content. This means that at this point we only handle:

```
<section>
  <title>Some title</title>
  <content>
    <p>a paragraph of text</p>
  </content>
</section>
```

In the next section we will deal with the slightly more complex itemize and figure placement. At first sight all these setups may look overkill but keep in mind that normally the number of elements is rather limited. The complexity is often in the style and having access to each snippet of content is actually quite handy for that.

<< 1.2 >> alternative solutions

Dealing with an itemize is rather simple (as long as we forget about attributes that control the behaviour):

```
<itemize>
  <item>first</item>
  <item>second</item>
</itemize>
```

First we need to add `itemize` to the setup assignment (unless we've used the wildcard `*`):

```
\xmlsetsetup{demo}{document|section|p|itemize}{xml:demo:*}
```

The setup can look like:

```
\startxmlsetups xml:demo:itemize
  \startitemize
    \xmlfilter{#1}{/item/command(xml:demo:itemize:item)}
  \stopitemize
\stopxmlsetups

\startxmlsetups xml:demo:itemize:item
  \startitem
    \xmlflush{#1}
  \stopitem
\stopxmlsetups
```

An alternative is to map item directly:

```
\xmlsetsetup{demo}{document|section|p|itemize|item}{xml:demo:*}
```

and use:

```
\startxmlsetups xml:demo:itemize
  \startitemize
    \xmlflush{#1}
  \stopitemize
\stopxmlsetups

\startxmlsetups xml:demo:item
  \startitem
    \xmlflush{#1}
  \stopitem
\stopxmlsetups
```

Sometimes, a more local solution using filters and `/command(...)` makes more sense, especially when the `item` tag is used for other purposes as well.

Explicit flushing with `command` is definitely the way to go when you have complex products. In one of our projects we compose math school books from many thousands of small XML files, and from one source set several products are typeset. Within a book sections get done differently, content gets used, ignored or interpreted differently depending on the kind of content, so there is a constant checking of attributes that drive the rendering. In that a generic setup for a title element makes less sense than explicit ones for each case. (We're talking of huge amounts of files here, including multiple images on each rendered page.)

When using `command` you can pass two arguments, the first is the setup for the match, the second one for the miss, as in:

```
\xmlfilter{#1}{/element/command(xml:true,xml:false)}
```

Back to the example, this leaves us with dealing with the resources, like figures:

```
<resource type='figure'>
  <caption>A picture of a cow.</caption>
  <content><external file="cow.pdf"/></content>
</resource>
```

Here we can use a more restricted match:

```
\xmlsetsetup{demo}{resource[@type='figure']}{xml:demo:figure}
\xmlsetsetup{demo}{external}{xml:demo:*}
```

and the definitions:

```
\startxmlsetups xml:demo:figure
```

```
  \placefigure
    {\xmlfirst{#1}{/caption}}
    {\xmlfirst{#1}{/content}}
\stopxmlsetups

\startxmlsetups xml:demo:external
  \externalfigure[\xmlatt{#1}{file}]
\stopxmlsetups
```

At this point it is good to notice that `\xmlatt{#1}{file}` is passed as it is: a macro call. This means that when a macro like `\externalfigure` uses the first argument frequently without first storing its value, the lookup is done several times. A solution for this is:

```
\startxmlsetups xml:demo:external
  \expanded{\externalfigure[\xmlatt{#1}{file}]}
\stopxmlsetups
```

Because the lookup is rather fast, normally there is no need to bother about this too much because internally ConTeXt already makes sure such expansion happens only once.

An alternative definition for placement is the following:

```
\xmlsetsetup{demo}{resource}{xml:demo:resource}
```

with:

```
\startxmlsetups xml:demo:resource
  \placefloat
    [\xmlatt{#1}{type}]
    {\xmlfirst{#1}{/caption}}
    {\xmlfirst{#1}{/content}}
\stopxmlsetups
```

This way you can specify `table` as type too. Because you can define your own float types, more complex variants are also possible. In that case it makes sense to provide some default behaviour too:

```
\definefloat[figure-here][figure][default=here]
\definefloat[figure-left][figure][default=left]
\definefloat[table-here] [table] [default=here]
\definefloat[table-left] [table] [default=left]

\startxmlsetups xml:demo:resource
  \placefloat
    [\xmlattdef{#1}{type}{figure}-\xmlattdef{#1}{location}{here}]
    {\xmlfirst{#1}{/caption}}
    {\xmlfirst{#1}{/content}}
\stopxmlsetups
```

In this example we support two types and two locations. We default to a figure placed (when possible) at the current location.

< 2 > Filtering content

<< 2.1 >> TEX versus LUA

It will not come as a surprise that we can access XML files from TEX as well as from LUA. In fact there are two methods to deal with XML in LUA. First there are the low level XML functions in the `xml` namespace. On top of those functions there is a set of functions in the `lxml` namespace that deals with XML in a more TEXie way. Most of these have similar commands at the TEX end.

```
\startxmlsetups first:demo:one
  \xmlfilter {#1} {artist/name[text()='Randy Newman']/..
    /albums/album[position()=3]/command(first:demo:two)}
\stopxmlsetups

\startxmlsetups first:demo:two
  \blank \start \tt
    \xmldisplayverbatim{#1}
  \stop \blank
\stopxmlsetups

\xmlprocessfile{demo}{music-collection.xml}{first:demo:one}
```

This gives the following snippet of verbatim XML code. The indentation is conform the indentation in the whole XML file.[1]

```
<name>Land Of Dreams</name>
<tracks>
 <track length="248">Dixie Flyer</track>
 <track length="212">New Orleans Wins The War</track>
 <track length="218">Four Eyes</track>
 <track length="181">Falling In Love</track>
 <track length="187">Something Special</track>
 <track length="168">Bad News From Home</track>
 <track length="207">Roll With The Punches</track>
 <track length="209">Masterman And Baby J</track>
 <track length="134">Follow The Flag</track>
 <track length="246">I Want You To Hurt Like I Do</track>
 <track length="248">It&apos;s Money That Matters</track>
 <track length="156">Red Bandana</track>
</tracks>
```

An alternative written in LUA looks as follows:

[1] The (probably outdated) XML file contains the collection stores on my slimserver instance. You can use the `mtxrun --script flac` to generate such files.

```
\blank \start \tt \startluacode
  local m = lxml.load("mine","music-collection.xml") -- m == lxml.id("mine")
  local p = "artist/name[text()='Randy Newman']/../albums/album[position()=4]"
  local l = lxml.filter(m,p) -- returns a list (with one entry)
  lxml.displayverbatim(l[1])
\stopluacode \stop \blank
```

This produces:

```
<name>Bad Love</name>
<tracks>
 <track length="340">My Country</track>
 <track length="295">Shame</track>
 <track length="205">I&apos;m Dead (But I Don&apos;t Know It)</track>
 <track length="213">Every Time It Rains</track>
 <track length="206">The Great Nations of Europe</track>
 <track length="220">The One You Love</track>
 <track length="164">The World Isn&apos;t Fair</track>
 <track length="264">Big Hat, No Cattle</track>
 <track length="243">Better Off Dead</track>
 <track length="236">I Miss You</track>
 <track length="126">Going Home</track>
 <track length="180">I Want Everyone To Like Me</track>
</tracks>
```

You can use both methods mixed but in practice we will use the TEX commands in regular styles and the mixture in modules, for instance in those dealing with MATHML and cals tables. For complex matters you can write your own finalizers (the last action to be taken in a match) in LUA and use them at the TEX end.

<< 2.2 >> a few details

In CONTEXT setups are a rather common variant on macros (TEX commands) but with their own namespace. An example of a setup is:

```
\startsetup doc:print
  \setuppapersize[A4][A4]
\stopsetup

\startsetup doc:screen
  \setuppapersize[S6][S4]
\stopsetup
```

Given the previous definitions, later on we can say something like:

```
\doifmodeelse {paper} {
```

```
  \setup[doc:print]
} {
  \setup[doc:screen]
}
```

Another example is:

```
\startsetup[doc:header]
  \marking[chapter]
  \space
  --
  \space
  \pagenumber
\stopsetup
```

in combination with:

```
\setupheadertexts[\setup{doc:header}]
```

Here the advantage is that instead of ending up with an unreadable header definitions, we use a nicely formatted setup. An important property of setups and the reason why they were introduced long ago is that spaces and newlines are ignored in the definition. This means that we don't have to worry about so called spurious spaces but it also means that when we do want a space, we have to use the `\space` command.

The only difference between setups and XML setups is that the following ones get an argument (`#1`) that reflects the current node in the XML tree.

<< 2.3 >> CDATA

What to do with `CDATA`? There are a few methods at tle LUA end for dealing with it but here we just mention how you can influence the rendering. There are four macros that play a role here:

```
\unexpanded\def\xmlcdataobeyedline {\obeyedline}
\unexpanded\def\xmlcdataobeyedspace{\strut\obeyedspace}
\unexpanded\def\xmlcdatabefore     {\begingroup\tt}
\unexpanded\def\xmlcdataafter      {\endgroup}
```

Technically you can overload them but beware of side effects. Normally you won't see much `CDATA` and whenever we do, it involves special data that needs very special treatment anyway.

<< 2.4 >> Entities

As usual with any way of encoding documents you need escapes in order to encode the characters that are used in tagging the content, embedding comments, escaping special characters in strings (in programming languages), etc. In XML this means that in order characters like `<` you need an escape like `<` and in order then to encode an `&` you need `&`.

In a typesetting workflow using a programming language like TEX, another problem shows up. There we have different special characters, like `$ $` for triggering math, but also the backslash, braces etc. Even one such special character is already enough to have yet another escaping mechanism at work.

Ideally a user should not worry about these issues but it helps figuring out issues when you know what happens under the hood. Also it is good to know that in the code there are several ways to deal with these issues. Take the following document:

```
<text>
    Here we have a bit of a &lt;&mess&gt;:

    # &#35;
    % &#37;
    \ &#92;
    { &#123;
    | &#124;
    } &#125;
    ~ &#126;
</text>
```

When the file is read the `<` entity will be replaced by `<` and the `>` by `>`. The numeric entities will be replaced by the characters they refer to. The `&mess` is kind of special. We do preload a huge list of more or less standardized entities but `mess` is not in there. However, it is possible to have it defined in the document preamble, like:

```
<!DOCTYPE dummy SYSTEM "dummy.dtd" [
    <!ENTITY mess "what a mess" >
]>
```

or even this:

```
<!DOCTYPE dummy SYSTEM "dummy.dtd" [
    <!ENTITY mess "<p>what a mess</p>" >
]>
```

You can also define it in your document style using one of:

`MISSING SETUP`

replaces entity with name `NAME` by `TEXT`

`MISSING SETUP`

replaces entity with name `NAME` by `TEXT` typeset under a TEX regime

Such a definition will always have a higher priority than the one defined in the document. Anyway, when the document is read in all entities are resolved and those that need a special treatment because

they map to some text are stored in such a way that we can roundtrip them. As a consequence, as soon as the content gets pushed into TEX, we need not only to intercept special characters but also have to make sure that the following works:

```
\xmltexentity {tex} {\TEX}
```

Here the backslash starts a control sequence while in regular content a backslash is just that: a backslash.

Special characters are really special when we have to move text around in a TEX ecosystem.

```
<text>
    <title>About #3</title>
</text>
```

If we map and define title as follows:

```
\startxmlsetup xml:title
    \title{\xmlflush{#1}}
\stopxmlsetup
```

normally something `\xmlflush{id::123}` will be written to the auxiliary file and in most cases that is quite okay, but if we have this:

```
\setuphead[title][expansion=yes]
```

then we don't want the `#` to end up as hash because later on TEX can get very confused about it because it sees some argument then in a probably unexpected way. This is solved by escaping the hash like this:

```
About \Ux{23}3
```

The `\Ux` command will convert its hexadecimal argument into a character. Of course one then needs to typeset such a text under a TEX character regime but that is normally the case anyway.

< 3 > Commands

<< 3.1 >> nodes and lpaths

The amount of commands available for manipulating the XML file is rather large. Many of the commands cooperate with the already discussed setups, a fancy name for a collection of macro calls either or not mixed with text.

Most of the commands are just shortcuts to LUA calls, which means that the real work is done by LUA. In fact, what happens is that we have a continuous transfer of control from TEX to LUA, where LUA prints back either data (like element content or attribute values) or just invokes a setup whereby it passes a reference to the node resolved conform the path expression. The invoked setup itself might return control to LUA again, etc.

This sounds complicated but examples will show what we mean here. First we present the whole repertoire of commands. Because users can read the source code, they might uncover more commands, but only the ones discussed here are official. The commands are grouped in categories.

In the following sections NODE means a reference to a node: this can be the identifier of the root (the loaded xml tree) or a reference to a node in that tree (often the result of some lookup. A LPATH is a fancy name for a path expression (as with XSLT) but resolved by LUA.

<< 3.2 >> commands

There are a lot of commands available but you probably can ignore most of them. We try to be complete which means that there is for instance `\xmlfirst` as well as `\xmllast` but you probably never need the last one. There are also commands that were used when testing this interface and we see no reason to remove them. Some obscure ones are used in modules and after a while even I often forget that they exist. To give you an idea of what commands are important we show their use in generating the CONTEXT command definitions (`x-set-11.mkiv`) per Januari 2016:

`\xmlall`	1	`\xmlflush`	5
`\xmlatt`	23	`\xmlinclude`	1
`\xmlattribute`	1	`\xmlloadonly`	1
`\xmlcount`	1	`\xmlregisterdocumentsetup`	1
`\xmldoif`	2	`\xmlsetsetup`	1
`\xmldoifelse`	1	`\xmlsetup`	4
`\xmlfilterlist`	4		

As you can see filtering, flushing and accessing attributes score high. Below we show the statistics of a quite complex rendering (5 variants of schoolbooks: basic book, answers, teachers guide, worksheets, full blown version with extensive tracing).

`\xmladdindex`	3	`\xmlappendsetup`	1
`\xmlall`	5	`\xmlapplyselectors`	1

\xmlatt	40	\xmlinclusion	16
\xmlattdef	9	\xmlinjector	1
\xmlattribute	10	\xmlloaddirectives	1
\xmlbadinclusions	3	\xmlmapvalue	4
\xmlconcat	3	\xmlmatch	1
\xmlcount	1	\xmlprependsetup	5
\xmldelete	11	\xmlregisterdocumentsetup	2
\xmldoif	39	\xmlregistersetup	1
\xmldoifelse	28	\xmlremapnamespace	1
\xmldoifelsetext	13	\xmlsetfunction	2
\xmldoifnot	2	\xmlsetinjectors	2
\xmldoifnotselfempty	1	\xmlsetsetup	11
\xmlfilter	100	\xmlsetup	76
\xmlfirst	51	\xmlstrip	1
\xmlflush	69	\xmlstripanywhere	1
\xmlflushcontext	2	\xmltag	1
\xmlinclude	1	\xmltext	53
\xmlincludeoptions	5	\xmlvalue	2

Here many more are used but this is an exceptional case. The top is again dominated by filtering, flushing and attribute consulting. The list can actually be smaller. For instance, the \xmlcount can just as well be \xmlfilter with a count finalizer. There are also some special ones, like the injectors, that are needed for finetuning the final result.

<< 3.3 >> loading

MISSING SETUP

loads the file FILE and registers it under NAME and applies either given or standard XMLSETUP (alias: \xmlload)

MISSING SETUP

loads the buffer BUFFER and registers it under NAME and applies either given or standard XMLSETUP

MISSING SETUP

loads TEXT and registers it under NAME and applies either given or standard XMLSETUP

MISSING SETUP

loads TEXT and registers it under NAME and applies either given or standard XMLSETUP but doesn't flush the content

MISSING SETUP

includes the file specified by attribute NAME of the element located by LPATH at node NODE

MISSING SETUP

registers file FILE as NAME and process the tree starting with XMLSETUP (alias: \xmlprocess)

MISSING SETUP

registers buffer NAME as NAME and process the tree starting with XMLSETUP

MISSING SETUP

registers TEXT as NAME and process the tree starting with XMLSETUP

The initial setup defaults to xml:process that is defined as follows:

```
\startsetups xml:process
  \xmlregistereddocumentsetups\xmldocument
  \xmlmain\xmldocument
\stopsetups
```

First we apply the setups associated with the document (including common setups) and then we flush the whole document. The macro \xmldocument expands to the current document id. There is also \xmlself which expands to the current node number (#1 in setups).

MISSING SETUP

returns the whole document

Normally such a flush will trigger a chain reaction of setups associated with the child elements.

<< 3.4 >> saving

MISSING SETUP

saves the given node NODE in the file FILE

MISSING SETUP

saves the match of LPATH in the file FILE

MISSING SETUP

saves the match of LPATH in the buffer BUFFER

MISSING SETUP

saves the match of LPATH verbatim in the buffer BUFFER

The next command is only needed when you have messed with the tree using LUA code.

MISSING SETUP

(re)indexes a tree

The following macros are only used in special situations and are not really meant for users.

MISSING SETUP

flush the content if NODE with original entities

MISSING SETUP

flush the wrapped content with original entities

<< 3.5 >> flushing data

When we flush an element, the associated XML setups are expanded. The most straightforward way to flush an element is the following. Keep in mind that the returned values itself can trigger setups and therefore flushes.

MISSING SETUP

returns all nodes under NODE

You can restrict flushing by using commands that accept a specification.

MISSING SETUP

returns the text of the matching LPATH under NODE

MISSING SETUP

returns the text of the matching LPATH under NODE without \Ux escaped special TEX characters

MISSING SETUP

returns the text of the NODE

MISSING SETUP

returns the text of the NODE without \Ux escaped special TEX characters

MISSING SETUP

returns the text of the matching LPATH under NODE without embedded spaces

MISSING SETUP

returns all nodes under NODE that matches LPATH

MISSING SETUP

returns all nodes found in the last match

MISSING SETUP

returns the first node under NODE that matches LPATH

MISSING SETUP

returns the last node under NODE that matches LPATH

MISSING SETUP

at a match of LPATH a given filter filter is applied and the result is returned

MISSING SETUP

returns the NUMBERth element under NODE

MISSING SETUP

returns the NUMBERth match of LPATH at node NODE; a negative number starts at the end (alias: \xmlindex)

MISSING SETUP

returns the NUMBERth child of node NODE; a negative number starts at the end

MISSING SETUP

returns the index (position) in the parent node of NODE

MISSING SETUP

returns the sequence of nodes that match LPATH at NODE whereby TEXT is put between each match

MISSING SETUP

returns the FIRSTth upto LASTth of nodes that match LPATH at NODE whereby TEXT is put between each match

MISSING SETUP

apply the given XMLSETUP to each match of LPATH at node NODE

MISSING SETUP

remove leading and trailing spaces from nodes under NODE that match LPATH

MISSING SETUP

remove leading and trailing spaces from nodes under NODE that match LPATH and return the content afterwards

MISSING SETUP

remove leading and trailing spaces as well as collapse embedded spaces from nodes under NODE that match LPATH

MISSING SETUP

remove leading and trailing spaces as well as collapse embedded spaces from nodes under NODE that match LPATH and return the content afterwards

MISSING SETUP

flushes the content verbatim code (without any wrapping, i.e. no fonts are selected and such)

MISSING SETUP

return the content of the node as inline verbatim code; no further interpretation (expansion) takes place and spaces are honoured; it uses the following wrapper

MISSING SETUP

wraps inline verbatim mode using the environment specified (a prefix xml: is added to the environment name)

MISSING SETUP

return the content of the node as display verbatim code; no further interpretation (expansion) takes place and leading and trailing spaces and newlines are treated special; it uses the following wrapper

MISSING SETUP

wraps the content in display verbatim using the environment specified (a prefix `xml:` is added to the environment name)

MISSING SETUP

pretty print (with colors) the node `NODE`; use the ConTeXt SciTE lexers when available (`\usemodule[scite]`)

MISSING SETUP

flush node `NODE` obeying spaces and newlines

MISSING SETUP

flush node `NODE` obeying newlines

<< 3.6 >> information

The following commands return strings. Normally these are used in tests.

MISSING SETUP

returns the complete name (including namespace prefix) of the given `NODE`

MISSING SETUP

returns the namespace of the given `NODE`

MISSING SETUP

returns the tag of the element, without namespace prefix

MISSING SETUP

returns the number of matches of `LPATH` at node `NODE`

MISSING SETUP

returns the value of attribute NAME or empty if no such attribute exists

MISSING SETUP

returns the value of attribute NAME or CD:STRING if no such attribute exists

MISSING SETUP

returns the value of attribute NAME or empty if no such attribute exists; a leading # is removed (nicer for tex)

MISSING SETUP

returns the value of attribute NAME or empty if no such attribute exists; backtracks till a match is found

MISSING SETUP

returns the value of attribute NAME or CD:STRING if no such attribute exists; backtracks till a match is found

MISSING SETUP

finds a first match for LPATH at NODE and returns the value of attribute NAME or empty if no such attribute exists

MISSING SETUP

finds a first match for LPATH at NODE and returns the value of attribute NAME or TEXT if no such attribute exists

MISSING SETUP

returns the last attribute found (this avoids a lookup)

<< 3.7 >> manipulation

You can use LUA code to manipulate the tree and it makes no sense to duplicate this in TEX. In the future we might provide an interface to some of this functionality. Keep in mind that manipuating the tree might have side effects as we maintain several indices into the tree that also needs to be updated then.

<< 3.8 >> integration

If you write a module that deals with XML, for instance processing cals tables, then you need ways to control specific behaviour. For instance, you might want to add a background to the table. Such directives are collected in XML files and can be loaded on demand.

MISSING SETUP

loads CONTEXT directives from FILE that will get interpreted when processing documents

A directives definition file looks as follows:

```
<?xml version="1.0" standalone="yes"?>

<directives>
  <directive attribute='id' value="100"
    setup="cdx:100"/>
  <directive attribute='id' value="101"
    setup="cdx:101"/>
  <directive attribute='cdx' value="colors" element="cals:table"
    setup="cdx:cals:table:colors"/>
  <directive attribute='cdx' value="vertical" element="cals:table"
    setup="cdx:cals:table:vertical"/>
  <directive attribute='cdx' value="noframe" element="cals:table"
    setup="cdx:cals:table:noframe"/>
  <directive attribute='cdx' value="*" element="cals:table"
    setup="cdx:cals:table:*"/>
</directives>
```

Examples of usage can be found in x-cals.mkiv. The directive is triggered by an attribute. Instead of a setup you can specify a setup to be applied before and after the node gets flushed.

MISSING SETUP

apply the setups directive associated with the node

MISSING SETUP

apply the before directives associated with the node

MISSING SETUP

apply the after directives associated with the node

MISSING SETUP

defines a directive that hooks into a handler

Normally a directive will be put in the XML file, for instance as:

```
<?context-mathml-directive minus reduction yes ?>
```

Here the `mathml` is the general class of directives and `minus` a subclass, in our case a specific element.

<< 3.9 >> setups

The basic building blocks of XML processing are setups. These are just collections of macros that are expanded. These setups get one argument passed (`#1`):

```
\startxmlsetups somedoc:somesetup
    \xmlflush{#1}
\stopxmlsetups
```

This argument is normally a number that internally refers to a specific node in the XML tree. The user should see it as an abstract reference and not depend on its numeric property. Just think of it as 'the current node'. You can (and probably will) call such setups using:

MISSING SETUP

expands setup `SETUP` and pass `NODE` as argument

However, in most cases the setups are associated to specific elements, something that users of XSLT might recognize as templates.

MISSING SETUP

associates function `LUAFUNCTION` to the elements in namespace `NAME` that match `LPATH`

MISSING SETUP

associates setups SETUP (TEX code) with the matching nodes of LPATH or root NODE

MISSING SETUP

pushes SETUP to the front of global list of setups

MISSING SETUP

adds SETUP to the global list of setups to be applied (alias: \xmlregistersetup)

MISSING SETUP

pushes SETUP into the global list of setups; the last setup is the position

MISSING SETUP

adds SETUP to the global list of setups; the last setup is the position

MISSING SETUP

removes SETUP from the global list of setups

MISSING SETUP

pushes SETUP to the front of list of setups to be applied to NAME

MISSING SETUP

adds SETUP to the list of setups to be applied to NAME (you can also use the alias: \xmlregisterdocumentsetup)

MISSING SETUP

pushes SETUP into the setups to be applied to NAME; the last setup is the position

MISSING SETUP

adds SETUP to the setups to be applied to NAME; the last setup is the position

MISSING SETUP

removes SETUP from the global list of setups to be applied to NAME

MISSING SETUP

removes all global setups

MISSING SETUP

removes all setups from the NAME specific list of setups to be applied

MISSING SETUP

setup
applies SETUP (can be a list) to NAME

MISSING SETUP

applies all global setups to the current document

MISSING SETUP

applies all document specific SETUP to document NAME

<< 3.10 >> testing

The following test macros all take a NODE as first argument and an LPATH as second:

MISSING SETUP

expands to TRUE when LPATH matches at node NODE

`MISSING SETUP`

expands to `TRUE` when `LPATH` does not match at node `NODE`

`MISSING SETUP`

expands to `TRUE` when `LPATH` matches at node `NODE` and to `FALSE` otherwise

`MISSING SETUP`

expands to `TRUE` when the node matching `LPATH` at node `NODE` has some content

`MISSING SETUP`

expands to `TRUE` when the node matching `LPATH` at node `NODE` has no content

`MISSING SETUP`

expands to `TRUE` when the node matching `LPATH` at node `NODE` has content and to `FALSE` otherwise

`MISSING SETUP`

expands to `TRUE` when the node matching `LPATH` at node `NODE` is empty and to `FALSE` otherwise

`MISSING SETUP`

expands to `TRUE` when the node is empty and to `FALSE` otherwise

`MISSING SETUP`

expands to `TRUE` when `NODE` is empty

`MISSING SETUP`

expands to `TRUE` when `NODE` is not empty

<< 3.11 >> initialization

The general setup command (not to be confused with setups) that deals with the MkIV tree handler is \setupxml. There are currently only a few options.

MISSING SETUP

When you set default to TEXT elements with no setup assigned will end up as text. When set to hidden such elements will be hidden. You can apply the default yourself using:

MISSING SETUP

presets the tree with root NODE to the handlers set up with \setupxml option default

You can set compress to yes in which case comment is stripped from the tree when the file is read.

MISSING SETUP

associates an internal namespace (like mml) with one given in the document as URL (like mathml)

MISSING SETUP

changes the namespace and tag of the matching elements

MISSING SETUP

replaces all references to the given namespace to a new one (applied recursively)

MISSING SETUP

sets the namespace of the matching elements unless a namespace is already set

<< 3.12 >> helpers

Often an attribute will determine the rendering and this may result in many tests. Especially when we have multiple attributes that control the output such tests can become rather extensive and redundant because one gets $n \times m$ or more such tests.

Therefore we have a convenient way to map attributes onto for instance strings or commands.

MISSING SETUP

associate a TEXT with a CATEGORY and NAME (alias: \xmlmapval)

MISSING SETUP

expand the value associated with a CATEGORY and NAME and if not resolved, expand to the TEXT (alias: \xmlval)

MISSING SETUP

associate a TEXT with a CATEGORY and NAME

This is used as follows. We define a couple of mappings in the same category:

```
\xmlmapvalue{emph}{bold}  {\bf}
\xmlmapvalue{emph}{italic}{\it}
```

Assuming that we have associated the following setup with the emph element, we can say (with #1 being the current element):

```
\startxmlsetups demo:emph
  \begingroup
    \xmlvalue{emph}{\xmlatt{#1}{type}}{}
  \endgroup
\stopxmlsetups
```

In this case we have no default. The type attribute triggers the actions, as in:

```
normal <emph type='bold'>bold</emph> normal
```

This mechanism is not really bound to elements and attributes so you can use this mechanism for other purposes as well.

< 4 > Expressions and filters

<< 4.1 >> path expressions

In the previous chapters we used LPATH expressions, which are a variant on xpath expressions as in XSLT but in this case more geared towards usage in TEX. This mechanisms will be extended when demands are there.

A path is a sequence of matches. A simple path expression is:

```
a/b/c/d
```

Here each / goes one level deeper. We can go backwards in a lookup with ..:

```
a/b/../d
```

We can also combine lookups, as in:

```
a/(b|c)/d
```

A negated lookup is preceded by a !:

```
a/(b|c)/!d
```

A wildcard is specified with a *:

```
a/(b|c)/!d/e/*/f
```

In addition to these tag based lookups we can use attributes:

```
a/(b|c)/!d/e/*/f[@type=whatever]
```

An @ as first character means that we are dealing with an attribute. Within the square brackets there can be boolean expressions:

```
a/(b|c)/!d/e/*/f[@type=whatever and @id>100]
```

You can use functions as in:

```
a/(b|c)/!d/e/*/f[something(text()) == "oeps"]
```

There are a couple of predefined functions:

rootposition order	number	the index of the matched root element (kind of special)
position	number	the current index of the matched element in the match list
match	number	the current index of the matched element sub list with the same parent
first	number	
last	number	

index	number	the current index of the matched element in its parent list
firstindex	number	
lastindex	number	
element	number	the element's index
firstelement	number	
lastelement	number	
text	string	the textual representation of the matched element
content	table	the node of the matched element
name	string	the full name of the matched element: namespace and tag
namespace ns	string	the namespace of the matched element
tag	string	the tag of the matched element
attribute	string	the value of the attribute with the given name of the matched element

There are fundamental differences between position, match and index. Each step results in a new list of matches. The position is the index in this new (possibly intermediate) list. The match is also an index in this list but related to the specific match of element names. The index refers to the location in the parent element.

Say that we have:

```
<collection>
  <resources>
    <manual>
      <screen>.1.</screen>
      <paper>.1.</paper>
    </manual>
    <manual>
      <paper>.2.</paper>
      <screen>.2.</screen>
    </manual>
  <resources>
  <resources>
    <manual>
      <screen>.3.</screen>
      <paper>.3.</paper>
    </manual>
  <resources>
<collection>
```

The following then applies:

```
collection/resources/manual[position()==1]/paper  .1.
collection/resources/manual[match()==1]/paper     .1. .3.
collection/resources/manual/paper[index()==1]     .2.
```

In most cases the position test is more restrictive than the match test.

You can pass your own functions too. Such functions are defined in the the `xml.expressions` namespace. We have defined a few shortcuts:

`find(str,pattern)`	`string.find`
`contains(str)`	`string.find`
`oneof(str,...)`	is `str` in list
`upper(str)`	`characters.upper`
`lower(str)`	`characters.lower`
`number(str)`	`tonumber`
`boolean(str)`	`toboolean`
`idstring(str)`	removes leading hash
`name(index)`	full tag name
`tag(index)`	tag name
`namespace(index)`	namespace of tag
`text(index)`	content
`error(str)`	quit and show error
`quit()`	quit
`print()`	print message
`count(pattern)`	number of matches
`child(pattern)`	take child that matches

You can also use normal LUA functions as long as you make sure that you pass the right arguments. There are a few predefined variables available inside such functions.

`list`	table	the list of matches
`l`	number	the current index in the list of matches
`ll`	element	the current element that matched
`order`	number	the position of the root of the path

The given expression between `[]` is converted to a LUA expression so you can use the usual operators:

```
== ~= <= >= < > not and or ()
```

In addition, `=` equals `==` and `!=` is the same as `~=`. If you mess up the expression, you quite likely get a LUA error message.

<< 4.2 >> functions as filters

At the LUA end a whole `LPATH` expression results in a (set of) node(s) with its environment, but that is hardly usable in TEX. Think of code like:

```
for e in xml.collected(xml.load('text.xml'),"title") do
  -- e = the element that matched
end
```

The older variant is still supported but you can best use the previous variant.

```
for r, d, k in xml.elements(xml.load('text.xml'),"title") do
```

```
  -- r = root of the title element
  -- d = data table
  -- k = index in data table
end
```

Here `d[k]` points to the `title` element and in this case all titles in the tree pass by. In practice this kind of code is encapsulated in function calls, like those returning elements one by one, or returning the first or last match. The result is then fed back into TEX, possibly after being altered by an associated setup. We've seen the wrappers to such functions already in a previous chapter.

In addition to the previously discussed expressions, one can add so called filters to the expression, for instance:

```
a/(b|c)/!d/e/text()
```

In a filter, the last part of the LPATH expression is a function call. The previous example returns the text of each element `e` that results from matching the expression. When running TEX the following functions are available. Some are also available when using pure LUA. In TEX you can often use one of the macros like `\xmlfirst` instead of a `\xmlfilter` with finalizer `first()`. The filter can be somewhat faster but that is hardly noticeable.

`context()`	string	the serialized text with TEX catcode regime
`function()`	string	depends on the function
`name()`	string	the (remapped) namespace
`tag()`	string	the name of the element
`tags()`	list	the names of the element
`text()`	string	the serialized text
`upper()`	string	the serialized text uppercased
`lower()`	string	the serialized text lowercased
`stripped()`	string	the serialized text stripped
`lettered()`	string	the serialized text only letters (cf. UNICODE)
`count()`	number	the number of matches
`index()`	number	the matched index in the current path
`match()`	number	the matched index in the preceding path
`attribute(name)`	content	returns the attribute with the given name
`chainattribute(name)`	content	sidem, but backtracks till one is found
`command(name)`	content	expands the setup with the given name for each found element
`position(n)`	content	processes the n^{th} instance of the found element
`all()`	content	processes all instances of the found element
`reverse()`	content	idem in reverse order
`first()`	content	processes the first instance of the found element
`last()`	content	processes the last instance of the found element
`concat(...)`	content	concatinates the match
`concatrange(from,to,...)`	content	concatinates a range of matches

The extra arguments of the concatinators are: `separator` (string), `lastseparator` (string) and `textonly` (a boolean).

These filters are in fact LUA functions which means that if needed more of them can be added. Indeed this happens in some of the XML related MKIV modules, for instance in the MATHML processor.

<< 4.3 >> example

The number of commands is rather large and if you want to avoid them this is often possible. Take for instance:

```
\xmlall{#1}{/a/b[position()>3]}
```

Alternatively you can use:

```
\xmlfilter{#1}{/a/b[position()>3]/all()}
```

and actually this is also faster as internally it avoids a function call. Of course in practice this is hardly measurable.

In previous examples we've already seen quite some expressions, and it might be good to point out that the syntax is modelled after XSLT but is not quite the same. The reason is that we started with a rather minimal system and have already styles in use that depend on compatibility.

```
namespace:// axis node(set) [expr 1]..[expr n] / ... / filter
```

When we are inside a CONTEXT run, the namespace is `tex`. Hoewever, if you want not to print back to TEX you need to be more explicit. Say that we typeset examns and have a (not that logical) structure like:

```
<question>
  <text>...</text>
  <answer>
    <item>one</item>
    <item>two</item>
    <item>three</item>
  </answer>
  <alternative>
    <condition>true</condition>
    <score>1</score>
  </alternative>
  <alternative>
    <condition>false</condition>
    <score>0</score>
  </alternative>
  <alternative>
    <condition>true</condition>
    <score>2</score>
  </alternative>
</question>
```

Say that we typeset the questions with:

```
\startxmlsetups question
  \blank
  score: \xmlfunction{#1}{totalscore}
  \blank
  \xmlfirst{#1}{text}
  \startitemize
      \xmlfilter{#1}{/answer/item/command(answer:item)}
  \stopitemize
  \endgraf
  \blank
\stopxmlsetups
```

Each item in the answer results in a call to:

```
\startxmlsetups answer:item
  \startitem
    \xmlflush{#1}
    \endgraf
    \xmlfilter{#1}{../../alternative[position()=rootposition()]/
      condition/command(answer:condition)}
  \stopitem
\stopxmlsetups

\startxmlsetups answer:condition
  \endgraf
  condition: \xmlflush{#1}
  \endgraf
\stopxmlsetups
```

Now, there are two rather special filters here. The first one involves calculating the total score. As we look forward we use a function to deal with this.

```
\startluacode
function xml.functions.totalscore(root)
  local score = 0
  for e in xml.collected(root,"/alternative") do
    score = score + xml.filter(e,"xml:///score/number()") or 0
  end
  tex.write(score)
end
\stopluacode
```

Watch how we use the namespace to keep the results at the LUA end.

The second special trick shown here is to limit a match using the current position of the root (#) match.

As you can see, a path expression can be more than just filtering a few nodes. At the end of this manual you will find a bunch of examples.

<< 4.4 >> tables

If you want to know how the internal XML tables look you can print such a table:

```
print(table.serialize(e))
```

This produces for instance:

```
t={
 ["at"]={
  ["label"]="whatever",
 },
 ["dt"]={ "some text" },
 ["ns"]="",
 ["rn"]="",
 ["tg"]="demo",
}
```

The `rn` entry is the renamed namespace (when renaming is applied). If you see tags like `@pi@` this means that we don't have an element, but (in this case) a processing instruction.

`@rt@`	the root element
`@dd@`	document definition
`@cm@`	comment, like `<!-- whatever -->`
`@cd@`	so called `CDATA`
`@pi@`	processing instruction, like `<?whatever we want ?>`

There are many ways to deal with the content, but in the perspective of TEX only a few matter.

`xml.sprint(e)`	print the content to TEX and apply setups if needed
`xml.tprint(e)`	print the content to TEX (serialize elements verbose)
`xml.cprint(e)`	print the content to TEX (used for special content)

Keep in mind that anything low level that you uncover is not part of the official interface unless mentioned in this manual.

< 5 > Tips and tricks

<< 5.1 >> tracing

It can be hard to debug code as much happens kind of behind the screens. Therefore we have a couple of tracing options. Of course you can typeset some status information, using for instance:

MISSING SETUP

typeset the tree given by NODE

MISSING SETUP

typeset the name if the element given by NODE

MISSING SETUP

returns the complete path (including namespace prefix and index) of the given NODE

Say that we have the following XML:

```
<?xml version "1.0"?>
<document>
    <section>
        <content>
            <p>first</p>
            <p><b>second</b></p>
        </content>
    </section>
    <section>
        <content>
            <p><b>third</b></p>
            <p>fourth</p>
        </content>
    </section>
</document>
```

and the next definitions:

```
\startxmlsetups xml:demo:base
    \xmlsetsetup{#1}{p|b}{xml:demo:*}
\stopxmlsetups
```

```
\startxmlsetups xml:demo:p
    \xmlflush{#1}
    \par
\stopxmlsetups

\startxmlsetups xml:demo:b
    \par
    \xmlpath{#1} : \xmlflush{#1}
    \par
\stopxmlsetups

\xmlregisterdocumentsetup{example-10}{xml:demo:base}

\xmlprocessbuffer{example-10}{demo}{}
```

This will give us:

first
document/section[1]/content[1]/p[2]/b[1] : second
document/section[2]/content[1]/p[1]/b[1] : third
fourth

If you use `\xmlshow` you will get a complete subtree which can be handy for tracing but can also lead to large documents.

We also have a bunch of trackers that can be enabled, like:

```
\enabletrackers[xml.show,xml.parse]
```

The full list (currently) is:

`xml.entities`	show what entities are seen and replaced
`xml.path`	show the result of parsing an lpath expression
`xml.parse`	show stepwise resolving of expressions
`xml.profile`	report all parsed lpath expressions (in the log)
`xml.remap`	show what namespaces are remapped
`lxml.access`	report errors with respect to resolving (symbolic) nodes
`lxml.comments`	show the comments that are encountered (if at all)
`lxml.loading`	show what files are loaded and converted
`lxml.setups`	show what setups are being associated to elements

In one of our workflows we produce books from XML where the (educational) content is organized in many small files. Each book has about 5 chapters and each chapter is made of sections that contain text, exercises, resources, etc. and so the document is assembled from thousands of files (don't worry, runtime inclusion is pretty fast). In order to see where in the sources content resides we can trace the filename.

MISSING SETUP

returns the file where the node comes from

MISSING SETUP

returns the list of files where the node comes from

MISSING SETUP

returns a list of files that were not included due to some problem

Of course you have to make sure that these names end up somewhere visible, for instance in the margin.

<< 5.2 >> expansion

For novice users the concept of expansion might sound frightening and to some extend it is. However, it is important enough to spend some words on it here.

It is good to realize that most setups are sort of immediate. When one setup is issued, it can call another one and so on. Normally you won't notice that but there are cases where that can be a problem. In TEX you can define a macro, take for instance:

```
\startxmlsetups xml:foo
  \def\foobar{\xmlfirst{#1}{/bar}}
\stopxmlsetups
```

you store the reference top node `bar` in `\foobar` maybe for later use. In this case the content is not yet fetched, it will be done when `\foobar` is called.

```
\startxmlsetups xml:foo
  \edef\foobar{\xmlfirst{#1}{/bar}}
\stopxmlsetups
```

Here the content of `bar` becomes the body of the macro. But what if `bar` itself contains elements that also contain elements. When there is a setup for `bar` it will be triggered and so on.

When that setup looks like:

```
\startxmlsetups xml:bar
  \def\barfoo{\xmlflush{#1}}
\stopxmlsetups
```

Here we get something like:

```
\foobar => {\def\barfoo{...}}
```

When \barfoo is not defined we get an error and when it is known and expands to something weird we might also get an error.

Especially when you don't know what content can show up, this can result in errors when an expansion fails, for example because some macro being used is not defined. To prevent this we can define a macro:

```
\starttexdefinition unexpanded xml:bar:macro #1
  \def\barfoo{\xmlflush{#1}}
\stoptexdefinition

\startxmlsetups xml:bar
  \texdefinition{xml:bar:macro}{#1}
\stopxmlsetups
```

The setup xml:bar will still expand but the replacement text now is just the call to the macro, think of:

```
\foobar => {\texdefinition{xml:bar:macro}{#1}}
```

But this is often not needed, most ConTeXt commands can handle the expansions quite well but it's good to know that there is a way out. So, now to some examples. Imagine that we have an XML file that looks as follows:

```
<?xml version='1.0' ?>
<demo>
    <chapter>
        <title>Some <em>short</em> title</title>
        <content>
            zeta
            <index>
                <key>zeta</key>
                <content>zeta again</content>
            </index>
            alpha
            <index>
                <key>alpha</key>
                <content>alpha <em>again</em></content>
            </index>
            gamma
            <index>
                <key>gamma</key>
                <content>gamma</content>
            </index>
            beta
```

```
            <index>
                <key>beta</key>
                <content>beta</content>
            </index>
            delta
            <index>
                <key>delta</key>
                <content>delta</content>
            </index>
            done!
        </content>
    </chapter>
</demo>
```

There are a few structure related elements here: a chapter (with its list entry) and some index entries. Both are multipass related and therefore travel around. This means that when we let data end up in the auxiliary file, we need to make sure that we end up with either expanded data (i.e. no references to the XML tree) or with robust forward and backward references to elements in the tree.

Here we discuss three approaches (and more may show up later): pushing XML into the auxiliary file and using references to elements either or not with an associated setup. We control the variants with a switch.

```
\newcount\TestMode

\TestMode=0 % expansion=xml
\TestMode=1 % expansion=yes, index, setup
\TestMode=2 % expansion=yes
```

We apply a couple of setups:

```
\startxmlsetups xml:mysetups
    \xmlsetsetup{\xmldocument}{demo|index|content|chapter|title|em}{xml:*}
\stopxmlsetups

\xmlregistersetup{xml:mysetups}
```

The main document is processed with:

```
\startxmlsetups xml:demo
    \xmlflush{#1}
    \subject{contents}
    \placelist[chapter][criterium=all]
    \subject{index}
    \placeregister[index][criterium=all]
    \page % else buffer is forgotten when placing header
\stopxmlsetups
```

First we show three alternative ways to deal with the chapter. The first case expands the XML reference so that we have an XML stream in the auxiliary file. This stream is processed as a small independent subfile when needed. The second case registers a reference to the current element (#1). This means that we have access to all data of this element, like attributes, title and content. What happens depends on the given setup. The third variant does the same but here the setup is part of the reference.

```
\startxmlsetups xml:chapter
    \ifcase \TestMode
        % xml code travels around
        \setuphead[chapter][expansion=xml]
        \startchapter[title=eh: \xmltext{#1}{title}]
            \xmlfirst{#1}{content}
        \stopchapter
    \or
        % index is used for access via setup
        \setuphead[chapter][expansion=yes,xmlsetup=xml:title:flush]
        \startchapter[title=\xmlgetindex{#1}]
            \xmlfirst{#1}{content}
        \stopchapter
    \or
        % tex call to xml using index is used
        \setuphead[chapter][expansion=yes]
        \startchapter[title=hm: \xmlreference{#1}{xml:title:flush}]
            \xmlfirst{#1}{content}
        \stopchapter
    \fi
\stopxmlsetups

\startxmlsetups xml:title:flush
    \xmltext{#1}{title}
\stopxmlsetups
```

We need to deal with emphasis and the content of the chapter.

```
\startxmlsetups xml:em
    \begingroup\em\xmlflush{#1}\endgroup
\stopxmlsetups

\startxmlsetups xml:content
    \xmlflush{#1}
\stopxmlsetups
```

A similar approach is followed with the index entries. Watch how we use the numbered entries variant (in this case we could also have used just `entries` and `keys`).

```
\startxmlsetups xml:index
```

```
    \ifcase \TestMode
        \setupregister[index][expansion=xml,xmlsetup=]
        \setstructurepageregister
          [index]
          [entries:1=\xmlfirst{#1}{content},
           keys:1=\xmltext{#1}{key}]
    \or
        \setupregister[index][expansion=yes,xmlsetup=xml:index:flush]
        \setstructurepageregister
          [index]
          [entries:1=\xmlgetindex{#1},
           keys:1=\xmltext{#1}{key}]
    \or
        \setupregister[index][expansion=yes,xmlsetup=]
        \setstructurepageregister
          [index]
          [entries:1=\xmlreference{#1}{xml:index:flush},
           keys:1=\xmltext{#1}{key}]
    \fi
\stopxmlsetups

\startxmlsetups xml:index:flush
    \xmlfirst{#1}{content}
\stopxmlsetups
```

Instead of this flush, you can use the predefined setup `xml:flush` unless it is overloaded by you.

The file is processed by:

```
\starttext
    \xmlprocessfile{main}{test.xml}{}
\stoptext
```

We don't show the result here. If you're curious what the output is, you can test it yourself. In that case it also makes sense to peek into the `test.tuc` file to see how the information travels around. The `metadata` fields carry information about how to process the data.

The first case, the XML expansion one, is somewhat special in the sense that internally we use small pseudo files. You can control the rendering by tweaking the following setups:

```
\startxmlsetups xml:ctx:sectionentry
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:ctx:registerentry
    \xmlflush{#1}
\stopxmlsetups
```

When these methods work out okay the other structural elements will be dealt with in a similar way.

<< 5.3 >> special cases

Normally the content will be flushed under a special (so called) catcode regime. This means that characters that have a special meaning in TEX will have no such meaning in an XML file. If you want content to be treated as TEX code, you can use one of the following:

MISSING SETUP

flush the given NODE using the TEX character interpretation scheme

MISSING SETUP

flush the match of LPATH for the given NODE using the TEX character interpretation scheme

We use this in cases like:

```
....
  \xmlsetsetup {#1} {
      tm|texformula|
  } {xml:*}
....

\startxmlsetups xml:tm
  \mathematics{\xmlflushcontext{#1}}
\stopxmlsetups

\startxmlsetups xml:texformula
  \placeformula\startformula\xmlflushcontext{#1}\stopformula
\stopxmlsetups
```

<< 5.4 >> collecting

Say that your document has

```
<table>
    <tr>
        <td>foo</td>
        <td>bar<td>
    </tr>
</table>
```

And that you need to convert that to TEX speak like:

```
\bTABLE
    \bTR
        \bTD foo \eTD
        \bTD bar \eTD
    \eTR
\eTABLE
```

A simple mapping is:

```
\startxmlsetups xml:table
    \bTABLE \xmlflush{#1} \eTABLE
\stopxmlsetups
\startxmlsetups xml:tr
    \bTR \xmlflush{#1} \eTR
\stopxmlsetups
\startxmlsetups xml:td
    \bTD \xmlflush{#1} \eTD
\stopxmlsetups
```

The \bTD command is a so called delimited command which means that it picks up its argument by looking for an \eTD. For the simple case here this works quite well because the flush is inside the pair. This is not the case in the following variant:

```
\startxmlsetups xml:td:start
    \bTD
\stopxmlsetups
\startxmlsetups xml:td:stop
    \eTD
\stopxmlsetups
\startxmlsetups xml:td
    \xmlsetup{#1}{xml:td:start}
    \xmlflush{#1}
    \xmlsetup{#1}{xml:td:stop}
\stopxmlsetups
```

When for some reason TEX gets confused you can revert to a mechanism that collects content.

```
\startxmlsetups xml:td:start
    \startcollect
        \bTD
    \stopcollect
\stopxmlsetups
\startxmlsetups xml:td:stop
    \startcollect
        \eTD
    \stopcollect
\stopxmlsetups
```

```
\startxmlsetups xml:td
    \startcollecting
        \xmlsetup{#1}{xml:td:start}
        \xmlflush{#1}
        \xmlsetup{#1}{xml:td:stop}
    \stopcollecting
\stopxmlsetups
```

You can even implement solutions that effectively do this:

```
\startcollecting
    \startcollect \bTABLE \stopcollect
        \startcollect \bTR \stopcollect
            \startcollect \bTD \stopcollect
            \startcollect   foo\stopcollect
            \startcollect \eTD \stopcollect
            \startcollect \bTD \stopcollect
            \startcollect   bar\stopcollect
            \startcollect \eTD \stopcollect
        \startcollect \eTR \stopcollect
    \startcollect \eTABLE \stopcollect
\stopcollecting
```

Of course you only need to go that complex when the situation demands it. Here is another weird one:

```
\startcollecting
    \startcollect \setupsomething[\stopcollect
        \startcollect foo=\stopcollect
        \startcollect FOO,\stopcollect
        \startcollect bar=\stopcollect
        \startcollect BAR,\stopcollect
    \startcollect ]\stopcollect
\stopcollecting
```

<< 5.5 >> selectors and injectors

This section describes a bit special feature, one that we needed for a project where we could not touch the original content but could add specific sections for our own purpose. Hopefully the example demonstrates its useability.

```
<?xml version="1.0" encoding="UTF-8"?>

<?context-directive message info 1: this is a demo file ?>
<?context-message-directive info 2: this is a demo file ?>

<one>
```

```
<two>
    <?context-select begin t1 t2 t3 ?>
        <three>
            t1 t2 t3
            <?context-directive injector crlf t1 ?>
            t1 t2 t3
        </three>
    <?context-select end ?>
    <?context-select begin t4 ?>
        <four>
            t4
        </four>
    <?context-select end ?>
    <?context-select begin t8 ?>
        <four>
            t8.0
            t8.0
        </four>
    <?context-select end ?>
    <?context-include begin t4 ?>
        <!--
            <three>
                t4.t3
                <?context-directive injector crlf t1 ?>
                t4.t3
            </three>
        -->
        <three>
            t3
            <?context-directive injector crlf t1 ?>
            t3
        </three>
    <?context-include end ?>
    <?context-select begin t8 ?>
        <four>
            t8.1
            t8.1
        </four>
    <?context-select end ?>
    <?context-select begin t8 ?>
        <four>
            t8.2
            t8.2
        </four>
    <?context-select end ?>
```

```
        <?context-select begin t4 ?>
            <four>
                t4
                t4
            </four>
        <?context-select end ?>
        <?context-directive injector page t7 t8 ?>
        foo
        <?context-directive injector blank t1 ?>
        bar
        <?context-directive injector page t7 t8 ?>
        bar
    </two>
</one>
```

First we show how to plug in a directive. Processing instructions like the following are normally ignored by an XML processor, unless they make sense to it.

```
<?context-directive message info 1: this is a demo file ?>
<?context-message-directive info 2: this is a demo file ?>
```

We can define a message handler as follows:

```
\def\MyMessage#1#2#3{\writestatus{#1}{#2 #3}}

\xmlinstalldirective{message}{MyMessage}
```

When this file is processed you will see this on the console:

```
info > 1: this is a demo file
info > 2: this is a demo file
```

The file has some sections that can be used or ignored. The recipe for obeying `t1` and `t4` is the following:

```
\xmlsetinjectors[t1]
\xmlsetinjectors[t4]

\startxmlsetups xml:initialize
    \xmlapplyselectors{#1}
    \xmlsetsetup {#1} {
        one|two|three|four
    } {xml:*}
\stopxmlsetups

\xmlregistersetup{xml:initialize}

\startxmlsetups xml:one
```

```
    [ONE \xmlflush{#1} ONE]
\stopxmlsetups

\startxmlsetups xml:two
    [TWO \xmlflush{#1} TWO]
\stopxmlsetups

\startxmlsetups xml:three
    [THREE \xmlflush{#1} THREE]
\stopxmlsetups

\startxmlsetups xml:four
    [FOUR \xmlflush{#1} FOUR]
\stopxmlsetups
```

This typesets:

[ONE [TWO [THREE t1 t2 t3 t1 t2 t3 THREE] [FOUR t4 FOUR] [THREE t4.t3 t4.t3 THREE] [THREE t3 t3 THREE] [FOUR t4 t4 FOUR] foo

bar bar TWO] ONE]

The include coding is kind of special: it permits adding content (in a comment) and ignoring the rest so that we indeed can add something without interfering with the original. Of course in a normal workflow such messy solutions are not needed, but alas, often workflows are not that clean, especially when one has no real control over the source.

MISSING SETUP

enables a list of injectors that will be used

MISSING SETUP

resets the list of injectors

MISSING SETUP

expands an injection (command); normally this one is only used (in some setup) or for testing

MISSING SETUP

analyze the tree NODE for marked sections that will be injected

We have some injections predefined:

```
\startsetups xml:directive:injector:page
    \page
\stopsetups

\startsetups xml:directive:injector:column
    \column
\stopsetups

\startsetups xml:directive:injector:blank
    \blank
\stopsetups
```

In the example we see:

```
<?context-directive injector page t7 t8 ?>
```

When we set `\xmlsetinjector[t7]` a pagebreak will injected in that spot. Tags like `t7`, `t8` etc. can represent versions.

<< 5.6 >> preprocessing

Say that you have the following XML setup:

```
\startxmlsetups pre:demo:initialize
    \xmlsetsetup{#1}{*}{pre:demo:*}
\stopxmlsetups

\xmlregisterdocumentsetup{pre:demo}{pre:demo:initialize}

\startxmlsetups pre:demo:root
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups pre:demo:bold
    \begingroup\bf\xmlflush{#1}\endgroup
\stopxmlsetups

\starttext
    \xmlprocessbuffer{pre:demo}{demo}{}
\stoptext
```

and that (such things happen) the input looks like this:

```
<root>
BAD TITLE: crap crap crap ...

BAD TITLE: crap crap crap ...
```

```
</root>
```

You can then clean up these `BAD TITLE`'s as follows:

```
\startluacode
    function lxml.preprocessor(data,settings)
        return string.find(data,"BAD TITLE:")
           and string.gsub(data,"BAD TITLE:","<bold>BAD TITLE:</bold>")
            or data
    end
\stopluacode
```

and get as result:

BAD TITLE: crap crap crap ...

BAD TITLE: crap crap crap ...

The preprocessor function gets as second argument the current settings, an d the field `currentresource` can be used to limit the actions to specific resources, in our case it's `buffer: demo`. Afterwards you can reset the proprocessor with:

Future versions might give some more control over preprocessors. For now consider it to be a quick hack.

< 6 > Lookups using lpaths

<< 6.1 >> introduction

There is not that much system in the following examples. They resulted from tests with different documents. The current implementation evolved out of the experimental code. For instance, I decided to add the multiple expressions in row handling after a few email exchanges with Jean-Michel Huffen.

One of the main differences between the way XSLT resolves a path and our way is the anchor. Take:

```
/something
something
```

The first one anchors in the current (!) element so it will only consider direct children. The second one does a deep lookup and looks at the descendants as well. Furthermore we have a few extra shortcuts like `**` in `a/**/b` which represents all descendants.

The expressions (between square brackets) has to be valid LUA and some preprocessing is done to resolve the built in functions. So, you might use code like:

```
my_lpeg_expression:match(text()) == "whatever"
```

given that `my_lpeg_expression` is known. In the examples below we use the visualizer to show the steps. Some are shown more than once as part of a set.

<< 6.2 >> special cases

```
pattern:

1  axis  auto-self
```

```
pattern: *

1  axis  auto-descendant
```

```
pattern: .

1  axis  self
```

```
pattern: /

1  axis  auto-self
```

<< 6.3 >> wildcards

```
pattern: *
```

```
1  axis  auto-descendant
```

```
pattern: *:*
```

```
1  axis  auto-descendant
```

```
pattern: /*
```

```
1  axis  child
```

```
pattern: /*:*
```

```
1  axis  auto-child
```

```
pattern: */*
```

```
1  axis  child
2  axis  child
```

```
pattern: *:*/*:*
```

```
1  axis   auto-descendant
2  nodes  *:*
3  axis   auto-child
```

```
pattern: a/*
```

```
1  axis   auto-descendant
2  nodes  *:a
3  axis   child
```

```
pattern: a/*:*
```

```
1  axis   auto-descendant
2  nodes  *:a
3  axis   auto-child
```

```
pattern: /a/*
```

```
1  axis   auto-child
2  nodes  *:a
3  axis   child
```

```
pattern: /a/*:*

1  axis   auto-child
2  nodes  *:a
3  axis   auto-child

pattern: /*

1  axis  child

pattern: /**

1  axis  descendant

pattern: /***

1  axis  descendant
```

<< 6.4 >> multiple steps

```
pattern: answer

1  axis   auto-descendant
2  nodes  *:answer

pattern: answer/test/*

1  axis   auto-descendant
2  nodes  *:answer
3  axis   auto-child
4  nodes  *:test
5  axis   child

pattern: answer/test/child::

1  axis   auto-descendant
2  nodes  *:answer
3  axis   auto-child
4  nodes  *:test
5  axis   child

pattern: answer/*

1  axis   auto-descendant
2  nodes  *:answer
3  axis   child
```

```
pattern: answer/*[tag()='p' and position()=1 and text()!='']
```

```
1  axis        auto-descendant
2  nodes       *:answer
3  axis        child
4  expression  tag()='p' and position()=1 and text()!=''
```

<< 6.5 >> pitfals

```
pattern: [oneof(lower(@encoding),'tex','context','ctx')]
```

```
1  axis        auto-descendant
2  expression  oneof(lower(@encoding),'tex','context','ctx')
```

```
pattern: .[oneof(lower(@encoding),'tex','context','ctx')]
```

```
1  axis        self
2  expression  oneof(lower(@encoding),'tex','context','ctx')
```

<< 6.6 >> more special cases

```
pattern: **
```

```
1  axis  descendant
```

```
pattern: *
```

```
1  axis  auto-descendant
```

```
pattern: ..
```

```
1  axis  parent
```

```
pattern: .
```

```
1  axis  self
```

```
pattern: //
```

```
1  axis  descendant-or-self
```

```
pattern: /
```

```
1  axis  auto-self
```

```
pattern: **/

1  axis  descendant

pattern: **/*

1  axis  descendant
2  axis  child

pattern: **/.

1  axis  descendant
2  axis  self

pattern: **//

1  axis  descendant
2  axis  descendant-or-self

pattern: */

1  axis  child

pattern: */*

1  axis  child
2  axis  child

pattern: */.

1  axis  child
2  axis  self

pattern: *//

1  axis  child
2  axis  descendant-or-self

pattern: /**/

1  axis  descendant

pattern: /**/*

1  axis  descendant
2  axis  child
```

```
pattern: /**/.

1  axis  descendant
2  axis  self


pattern: /**//

1  axis  descendant
2  axis  descendant-or-self


pattern: /*/

1  axis  child


pattern: /*/*

1  axis  child
2  axis  child


pattern: /*/.

1  axis  child
2  axis  self


pattern: /*//

1  axis  child
2  axis  descendant-or-self


pattern: ./

1  axis  self


pattern: ./*

1  axis  self
2  axis  child


pattern: ./.

1  axis  self
2  axis  self
```

```
pattern: .//
```

```
1 axis  self
2 axis  descendant-or-self
```

```
pattern: ../
```

```
1 axis  parent
```

```
pattern: ../*
```

```
1 axis  parent
2 axis  child
```

```
pattern: ../.
```

```
1 axis  parent
2 axis  self
```

```
pattern: ..//
```

```
1 axis  parent
2 axis  descendant-or-self
```

<< 6.7 >> more wildcards

```
pattern: one//two
```

```
1 axis   auto-descendant
2 nodes  *:one
3 axis   descendant-or-self
4 nodes  *:two
```

```
pattern: one/*/two
```

```
1 axis   auto-descendant
2 nodes  *:one
3 axis   child
4 axis   auto-child
5 nodes  *:two
```

```
pattern: one/**/two
```

```
1 axis   auto-descendant
2 nodes  *:one
```

```
3  axis   descendant
4  axis   auto-child
5  nodes  *:two
```

```
pattern: one/***/two
```

```
1  axis   auto-descendant
2  nodes  *:one
3  axis   descendant-or-self
4  nodes  *:two
```

```
pattern: one/x//two
```

```
1  axis   auto-descendant
2  nodes  *:one
3  axis   auto-child
4  nodes  *:x
5  axis   descendant-or-self
6  nodes  *:two
```

```
pattern: one//x/two
```

```
1  axis   auto-descendant
2  nodes  *:one
3  axis   descendant-or-self
4  nodes  *:x
5  axis   auto-child
6  nodes  *:two
```

```
pattern: //x/two
```

```
1  axis   descendant-or-self
2  nodes  *:x
3  axis   auto-child
4  nodes  *:two
```

<< 6.8 >> special axis

```
pattern: descendant::whocares/ancestor::whoknows
```

```
1  axis   descendant
2  nodes  *:whocares
3  axis   ancestor
4  nodes  *:whoknows
```

```
pattern:  descendant::whocares/ancestor::whoknows/parent::

1  axis   descendant
2  nodes  *:whocares
3  axis   ancestor
4  nodes  *:whoknows
5  axis   parent


pattern:  descendant::whocares/ancestor::

1  axis   descendant
2  nodes  *:whocares
3  axis   ancestor


pattern:  child::something/child::whatever/child::whocares

1  axis   child
2  nodes  *:something
3  axis   child
4  nodes  *:whatever
5  axis   child
6  nodes  *:whocares


pattern:  child::something/child::whatever/child::whocares|whoknows

1  axis   child
2  nodes  *:something
3  axis   child
4  nodes  *:whatever
5  axis   child
6  nodes  *:whocares|*:whoknows


pattern:  child::something/child::whatever/child::(whocares|whoknows)

1  axis   child
2  nodes  *:something
3  axis   child
4  nodes  *:whatever
5  axis   child
6  nodes  *:whocares|*:whoknows


pattern:  child::something/child::whatever/child::!(whocares|whoknows)

1  axis   child
2  nodes  *:something
```

```
3  axis   child
4  nodes  *:whatever
5  axis   child
6  nodes  not(*:whocares|*:whoknows)
```

```
pattern:  child::something/child::whatever/child::(whocares)
```

```
1  axis   child
2  nodes  *:something
3  axis   child
4  nodes  *:whatever
5  axis   child
6  nodes  *:whocares
```

```
pattern:  child::something/child::whatever/child::(whocares)[position()>2]
```

```
1  axis        child
2  nodes       *:something
3  axis        child
4  nodes       *:whatever
5  axis        child
6  nodes       *:whocares
7  expression  position()>2
```

```
pattern:  child::something/child::whatever[position()>2][position()=1]
```

```
1  axis        child
2  nodes       *:something
3  axis        child
4  nodes       *:whatever
5  expression  position()>2
6  expression  position()=1
```

```
pattern:  child::something/child::whatever[whocares][whocaresnot]
```

```
1  axis        child
2  nodes       *:something
3  axis        child
4  nodes       *:whatever
5  expression  whocares
6  expression  whocaresnot
```

```
pattern:  child::something/child::whatever[whocares][not(whocaresnot)]
```

```
1  axis        child
2  nodes       *:something
```

```
3  axis        child
4  nodes       *:whatever
5  expression  whocares
6  expression  not(whocaresnot)
```

```
pattern: child::something/child::whatever/self::whatever
```

```
1  axis   child
2  nodes  *:something
3  axis   child
4  nodes  *:whatever
5  axis   self
6  nodes  *:whatever
```

There is also `last-match::` that starts with the last found set of nodes. This can save some runtime when you do lots of tests combined with a same check afterwards.

<< 6.9 >> some more examples

```
pattern: /something/whatever
```

```
1  axis   auto-child
2  nodes  *:something
3  axis   auto-child
4  nodes  *:whatever
```

```
pattern: something/whatever
```

```
1  axis   auto-descendant
2  nodes  *:something
3  axis   auto-child
4  nodes  *:whatever
```

```
pattern: /**/whocares
```

```
1  axis   descendant
2  axis   auto-child
3  nodes  *:whocares
```

```
pattern: whoknows/whocares
```

```
1  axis   auto-descendant
2  nodes  *:whoknows
3  axis   auto-child
4  nodes  *:whocares
```

```
pattern: whoknows
```

```
1 axis   auto-descendant
2 nodes  *:whoknows
```

```
pattern: whocares[contains(text(),'f') or contains(text(),'g')]
```

```
1 axis        auto-descendant
2 nodes       *:whocares
3 expression  contains(text(),'f') or contains(text(),'g')
```

```
pattern: whocares/first()
```

```
1 axis       auto-descendant
2 nodes      *:whocares
3 finalizer  first()
```

```
pattern: whocares/last()
```

```
1 axis       auto-descendant
2 nodes      *:whocares
3 finalizer  last()
```

```
pattern: whatever/all()
```

```
1 axis       auto-descendant
2 nodes      *:whatever
3 finalizer  all()
```

```
pattern: whocares/position(2)
```

```
1 axis       auto-descendant
2 nodes      *:whocares
3 finalizer  position("2")
```

```
pattern: whocares/position(-2)
```

```
1 axis       auto-descendant
2 nodes      *:whocares
3 finalizer  position("-2")
```

```
pattern: whocares[1]
```

```
1 axis        auto-descendant
2 nodes       *:whocares
3 expression  1
```

```
pattern: whocares[-1]

1  axis        auto-descendant
2  nodes       *:whocares
3  expression  -1


pattern: whocares[2]

1  axis        auto-descendant
2  nodes       *:whocares
3  expression  2


pattern: whocares[-2]

1  axis        auto-descendant
2  nodes       *:whocares
3  expression  -2


pattern: whatever[3]/attribute(id)

1  axis        auto-descendant
2  nodes       *:whatever
3  expression  3
4  finalizer   attribute("id")


pattern: whatever[2]/attribute('id')

1  axis        auto-descendant
2  nodes       *:whatever
3  expression  2
4  finalizer   attribute('id')


pattern: whatever[3]/text()

1  axis        auto-descendant
2  nodes       *:whatever
3  expression  3
4  finalizer   text()


pattern: /whocares/first()

1  axis       auto-child
2  nodes      *:whocares
3  finalizer  first()
```

```
pattern: /whocares/last()

1 axis       auto-child
2 nodes      *:whocares
3 finalizer  last()

pattern: xml://whatever/all()

1 axis       auto-descendant
2 nodes      *:whatever
3 finalizer  all()

pattern: whatever/all()

1 axis       auto-descendant
2 nodes      *:whatever
3 finalizer  all()

pattern: //whocares

1 axis   descendant-or-self
2 nodes  *:whocares

pattern: ..[2]

1 axis       parent
2 expression 2

pattern: ../*[2]

1 axis       parent
2 axis       child
3 expression 2

pattern: /(whocares|whocaresnot)

1 axis   auto-child
2 nodes  *:whocares|*:whocaresnot

pattern: /!(whocares|whocaresnot)

1 axis   auto-child
2 nodes  not(*:whocares|*:whocaresnot)
```

```
pattern: /!whocares
```

```
1 axis   auto-child
2 nodes  not(*:whocares)
```

```
pattern: /interface/command/command(xml:setups:register)
```

```
1 axis       auto-child
2 nodes      *:interface
3 axis       auto-child
4 nodes      *:command
5 finalizer  command("xml:setups:register")
```

```
pattern: /interface/command[@name='xxx']/command(xml:setups:typeset)
```

```
1 axis        auto-child
2 nodes       *:interface
3 axis        auto-child
4 nodes       *:command
5 expression  @name='xxx'
6 finalizer   command("xml:setups:typeset")
```

```
pattern: /arguments/*
```

```
1 axis   auto-child
2 nodes  *:arguments
3 axis   child
```

```
pattern: /sequence/first()
```

```
1 axis       auto-child
2 nodes      *:sequence
3 finalizer  first()
```

```
pattern: /arguments/text()
```

```
1 axis       auto-child
2 nodes      *:arguments
3 finalizer  text()
```

```
pattern: /sequence/variable/first()
```

```
1 axis       auto-child
2 nodes      *:sequence
3 axis       auto-child
```

```
4  nodes      *:variable
5  finalizer  first()
```

```
pattern: /interface/define[@name='xxx']/first()
```

```
1  axis        auto-child
2  nodes       *:interface
3  axis        auto-child
4  nodes       *:define
5  expression  @name='xxx'
6  finalizer   first()
```

```
pattern: /parameter/command(xml:setups:parameter:measure)
```

```
1  axis       auto-child
2  nodes      *:parameter
3  finalizer  command("xml:setups:parameter:measure")
```

```
pattern: /(*:library|figurelibrary)/*:figure/*:label
```

```
1  axis   auto-child
2  nodes  *:library|*:figurelibrary
3  axis   auto-child
4  nodes  *:figure
5  axis   auto-child
6  nodes  *:label
```

```
pattern: /(*:library|figurelibrary)/figure/*:label
```

```
1  axis   auto-child
2  nodes  *:library|*:figurelibrary
3  axis   auto-child
4  nodes  *:figure
5  axis   auto-child
6  nodes  *:label
```

```
pattern: /(*:library|figurelibrary)/figure/label
```

```
1  axis   auto-child
2  nodes  *:library|*:figurelibrary
3  axis   auto-child
4  nodes  *:figure
5  axis   auto-child
6  nodes  *:label
```

```
pattern: /(*:library|figurelibrary)/figure:*/label

1  axis   auto-child
2  nodes  *:library|*:figurelibrary
3  axis   auto-child
4  nodes  figure:*
5  axis   auto-child
6  nodes  *:label
```

< 7 > Examples

<< 7.1 >> attribute chains

In CSS, when an attribute is not present, the parent element is checked, and when not found again, the lookup follows the chain till a match is found or the root is reached. The following example demonstrates how such a chain lookup works.

```
<something mine="1" test="one" more="alpha">
  <whatever mine="2" test="two">
    <whocares mine="3">
      <!-- this is a test -->
    </whocares>
  </whatever>
</something>
```

We apply the following setups to this tree:

```
\startxmlsetups xml:common
    [
        \xmlchainatt{#1}{mine},
        \xmlchainatt{#1}{test},
        \xmlchainatt{#1}{more},
        \xmlchainatt{#1}{none}
    ]\par
\stopxmlsetups

\startxmlsetups xml:something
    something: \xmlsetup{#1}{xml:common}
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:whatever
    whatever: \xmlsetup{#1}{xml:common}
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:whocares
    whocares: \xmlsetup{#1}{xml:common}
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:mysetups
    \xmlsetsetup{#1}{something|whatever|whocares}{xml:*}
\stopxmlsetups
```

```
\xmlregisterdocumentsetup{example-1}{xml:mysetups}

\xmlprocessbuffer{example-1}{test}{}
```

This gives:

something: [1,one,alpha,]

whatever: [21,twoone,alpha,]

whocares: [321,twoone,alpha,]

<< 7.2 >> conditional setups

Say that we have this code:

```
\xmldoifelse {#1} {/what[@a='1']} {
    \xmlfilter {#1} {/what/command('xml:yes')}
} {
    \xmlfilter {#1} {/what/command('xml:nop')}
}
```

Here we first determine if there is a child `what` with attribute `a` set to `1`. Depending on the outcome again we check the child nodes for being named `what`. A faster solution which also takes less code is this:

```
\xmlfilter {#1} {/what[@a='1']/command('xml:yes','xml:nop')}
```

<< 7.3 >> manipulating

Assume that we have the following XML data:

```
<A>
  <B>right</B>
  <B>wrong</B>
</A>
```

But, instead of `right` we want to see `okay`. We can do that with a finalizer:

```
\startluacode
local rehash = {
    ["right"] = "okay",
}

function xml.finalizers.tex.Okayed(collected,what)
    for i=1,#collected do
        if what == "all" then
            local str = xml.text(collected[i])
```

```
            context(rehash[str] or str)
        else
            context(str)
        end
    end
end
\stopluacode

\startxmlsetups xml:A
       \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:B
    (It's \xmlfilter{#1}{./Okayed("all")})
\stopxmlsetups

\startxmlsetups xml:testsetups
       \xmlsetsetup{#1}{A|B}{xml:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-2}{xml:testsetups}
\xmlprocessbuffer{example-2}{test}{}
```

The result is:

(It's okay) (It's wrong)

<< 7.4 >> cross referencing

A rather common way to add cross references to XML files is to borrow the asymmetrical id's from HTML. This means that one cannot simply use a value of (say) `href` to locate an `id`. The next example came up on the ConTeXt mailing list.

```
<doc>
    <p>Text
        <a href="#fn1" class="footnoteref" id="fnref1"><sup>1</sup></a> and
        <a href="#fn2" class="footnoteref" id="fnref2"><sup>2</sup></a>
    </p>
    <div class="footnotes">
        <hr />
        <ol>
            <li id="fn1"><p>A footnote.<a href="#fnref1"></a></p></li>
            <li id="fn2"><p>A second footnote.<a href="#fnref2"></a></p></li>
        </ol>
    </div>
</doc>
```

We give two variants for dealing with such references. The first solution does lookups and depending on the size of the file can be somewhat inefficient.

```
\startxmlsetups xml:doc
    \blank
    \xmlflush{#1}
    \blank
\stopxmlsetups

\startxmlsetups xml:p
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:footnote
    (variant 1)\footnote
        {\xmlfirst
            {example-3-1}
            {div[@class='footnotes']/ol/li[@id='\xmlrefatt{#1}{href}']}}
\stopxmlsetups

\startxmlsetups xml:initialize
    \xmlsetsetup{#1}{p|doc}{xml:*}
    \xmlsetsetup{#1}{a[@class='footnoteref']}{xml:footnote}
    \xmlsetsetup{#1}{div[@class='footnotes']}{xml:nothing}
\stopxmlsetups

\xmlresetdocumentsetups{*}
\xmlregisterdocumentsetup{example-3-1}{xml:initialize}

\xmlprocessbuffer{example-3-1}{test}{}
```

This will typeset two footnotes.

Text (variant 1)[2] and (variant 1)[3]

The second variant collects the references so that the time spend on lookups is less.

```
\startxmlsetups xml:doc
    \blank
    \xmlflush{#1}
    \blank
\stopxmlsetups

\startxmlsetups xml:p
    \xmlflush{#1}
```

[2] A footnote.
[3] A second footnote.

```
\stopxmlsetups

\startluacode
    userdata.notes =  {}
\stopluacode

\startxmlsetups xml:collectnotes
    \ctxlua{userdata.notes['\xmlrefatt{#1}{id}'] = '#1'}
\stopxmlsetups

\startxmlsetups xml:footnote
    (variant 2)\footnote
        {\xmlflush
            {\cldcontext{userdata.notes['\xmlrefatt{#1}{href}']}}}
\stopxmlsetups

\startxmlsetups xml:initialize
    \xmlsetsetup{#1}{p|doc}{xml:*}
    \xmlsetsetup{#1}{a[@class='footnoteref']}{xml:footnote}
    \xmlfilter{#1}{div[@class='footnotes']/ol/li/command(xml:collectnotes)}
    \xmlsetsetup{#1}{div[@class='footnotes']}{}
\stopxmlsetups

\xmlregisterdocumentsetup{example-3-2}{xml:initialize}

\xmlprocessbuffer{example-3-2}{test}{}
```

This will again typeset two footnotes:

Text (variant 2)[4] and (variant 2)[5]

<< 7.5 >> mapping values

One way to process options `frame` in the example below is to map the values to values known by ConTeXt.

```
<a>
  <nattable frame="on">
    <tr><td>#1</td><td>#2</td><td>#3</td><td>#4</td></tr>
    <tr><td>#5</td><td>#6</td><td>#7</td><td>#8</td></tr>
  </nattable>
  <nattable frame="off">
    <tr><td>#1</td><td>#2</td><td>#3</td><td>#4</td></tr>
    <tr><td>#5</td><td>#6</td><td>#7</td><td>#8</td></tr>
```

[4] A footnote.
[5] A second footnote.

```
  </nattable>
  <nattable frame="no">
    <tr><td>#1</td><td>#2</td><td>#3</td><td>#4</td></tr>
    <tr><td>#5</td><td>#6</td><td>#7</td><td>#8</td></tr>
  </nattable>
</a>
```

The \xmlmapvalue mechanism is rather efficient and involves a minimum of testing.

```
\startxmlsetups xml:a
    \xmlflush{#1}
\stopxmlsetups

\xmlmapvalue {nattable:frame} {on}  {on}
\xmlmapvalue {nattable:frame} {yes} {on}
\xmlmapvalue {nattable:frame} {off} {off}
\xmlmapvalue {nattable:frame} {no}  {off}

\startxmlsetups xml:nattable
    \startplacetable[title=#1]
        \setupTABLE[frame=\xmlval{nattable:frame}{\xmlatt{#1}{frame}}{on}]%
        \bTABLE
            \xmlflush{#1}
        \eTABLE
    \stopplacetable
\stopxmlsetups

\startxmlsetups xml:tr
    \bTR
        \xmlflush{#1}
    \eTR
\stopxmlsetups

\startxmlsetups xml:td
    \bTD
        \xmlflush{#1}
    \eTD
\stopxmlsetups

\startxmlsetups xml:testsetups
    \xmlsetsetup{example-4}{a|nattable|tr|td|}{xml:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-4}{xml:testsetups}

\xmlprocessbuffer{example-4}{test}{}
```

We get:

#1	#2	#3	#4
#5	#6	#7	#8

Table 7.1 example-4::3

#1 #2 #3 #4

#5 #6 #7 #8

Table 7.2 example-4::14

<< 7.6 >> using LUA

In this example we demonstrate how you can delegate rendering to LUA. We will construct a so called extreme table. The input is:

```
<?xml version="1.0" encoding="utf-8"?>

<a>
  <b> <c>1</c> <d>Text</d>           </b>
  <b> <c>2</c> <d>More text</d>      </b>
  <b> <c>2</c> <d>Even more text</d> </b>
  <b> <c>2</c> <d>And more</d>       </b>
  <b> <c>3</c> <d>And even more</d>  </b>
  <b> <c>2</c> <d>The last text</d>  </b>
</a>
```

The processor code is:

```
\startxmlsetups xml:a
    \xmlflush{#1}
\stopxmlsetups

\xmlmapvalue {nattable:frame} {on}  {on}
\xmlmapvalue {nattable:frame} {yes} {on}
\xmlmapvalue {nattable:frame} {off} {off}
\xmlmapvalue {nattable:frame} {no}  {off}

\startxmlsetups xml:nattable
    \startplacetable[title=#1]
        \setupTABLE[frame=\xmlval{nattable:frame}{\xmlatt{#1}{frame}}{on}]%
        \bTABLE
            \xmlflush{#1}
        \eTABLE
    \stopplacetable
\stopxmlsetups

\startxmlsetups xml:tr
```

#1 #2 #3 #4

#5 #6 #7 #8

Table 7.3 example-4::25

```
    \bTR
        \xmlflush{#1}
    \eTR
\stopxmlsetups

\startxmlsetups xml:td
    \bTD
        \xmlflush{#1}
    \eTD
\stopxmlsetups

\startxmlsetups xml:testsetups
    \xmlsetsetup{example-4}{a|nattable|tr|td|}{xml:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-4}{xml:testsetups}

\xmlprocessbuffer{example-4}{test}{}
```

We color a sequence of the same titles (numbers here) differently. The first solution remembers the last title:

```
\startxmlsetups xml:a
    \startembeddedxtable
        \xmlflush{#1}
    \stopembeddedxtable
\stopxmlsetups

\startxmlsetups xml:b
    \xmlfunction{#1}{test_ba}
\stopxmlsetups

\startluacode
local lasttitle = nil

function xml.functions.test_ba(t)
    local title   = xml.text(t, "/c")
    local content = xml.text(t, "/d")
    context.startxrow()
    context.startxcell {
        background      = "color",
        backgroundcolor = lasttitle == title and "colorone" or "colortwo",
```

```
        foregroundstyle = "bold",
        foregroundcolor = "white",
    }
    context(title)
    lasttitle = title
    context.stopxcell()
    context.startxcell()
    context(content)
    context.stopxcell()
    context.stopxrow()
end
\stopluacode
```

The `embeddedxtable` environment is needed because the table is picked up as argument.

1	Text
2	More text
2	Even more text
2	And more
3	And even more
2	The last text

The second implemetation remembers what titles are already processed so here we can color the last one too.

```
\startxmlsetups xml:a
    \ctxlua{xml.functions.reset_bb()}
    \startembeddedxtable
        \xmlflush{#1}
    \stopembeddedxtable
\stopxmlsetups

\startxmlsetups xml:b
    \xmlfunction{#1}{test_bb}
\stopxmlsetups

\startluacode
local titles

function xml.functions.reset_bb(t)
    titles = { }
end

function xml.functions.test_bb(t)
    local title   = xml.text(t, "/c")
```

```
    local content = xml.text(t, "/d")
    context.startxrow()
    context.startxcell {
        background      = "color",
        backgroundcolor = titles[title] and "colorone" or "colortwo",
        foregroundstyle = "bold",
        foregroundcolor = "white",
    }
    context(title)
    titles[title] = true
    context.stopxcell()
    context.startxcell()
    context(content)
    context.stopxcell()
    context.stopxrow()
end
\stopluacode
```

1	Text
2	More text
2	Even more text
2	And more
3	And even more
2	The last text

A solution without any state variable is given below.

```
\startxmlsetups xml:a
    \startembeddedxtable
        \xmlflush{#1}
    \stopembeddedxtable
\stopxmlsetups

\startxmlsetups xml:b
    \xmlfunction{#1}{test_bc}
\stopxmlsetups

\startluacode
function xml.functions.test_bc(t)
    local title   = xml.text(t, "/c")
    local content = xml.text(t, "/d")
    context.startxrow()
    local okay = xml.text(t,"./preceding-sibling::/[-1]") == title
    context.startxcell {
        background      = "color",
```

```
        backgroundcolor = okay and "colorone" or "colortwo",
        foregroundstyle = "bold",
        foregroundcolor = "white",
    }
    context(title)
    context.stopxcell()
    context.startxcell()
    context(content)
    context.stopxcell()
    context.stopxrow()
end
\stopluacode
```

1	Text
2	More text
2	Even more text
2	And more
3	And even more
2	The last text

Here is a solution that delegates even more to LUA. The previous variants were actually not that safe with repect to special characters and didn't handle nested elements either but the next one does.

```
<?xml version="1.0" encoding="utf-8"?>

<a>
  <b> <c>#1</c> <d>Text</d>                      </b>
  <b> <c>#2</c> <d>More text</d>                 </b>
  <b> <c>#2</c> <d>Even more text</d>            </b>
  <b> <c>#2</c> <d>And more</d>                  </b>
  <b> <c>#3</c> <d>And even more</d>             </b>
  <b> <c>#2</c> <d>Something <i>nested</i> </d> </b>
</a>
```

We also need to map the `i` element.

```
\startxmlsetups xml:a
    \starttexcode
        \xmlfunction{#1}{test_a}
    \stoptexcode
\stopxmlsetups

\startxmlsetups xml:c
    \xmlflush{#1}
\stopxmlsetups
```

```
\startxmlsetups xml:d
    \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:i
    {\em\xmlflush{#1}}
\stopxmlsetups

\startluacode
function xml.functions.test_a(t)
    context.startxtable()
    local previous = false
    for b in xml.collected(lxml.getid(t),"/b") do
        context.startxrow()
            local current = xml.text(b,"/c")
            context.startxcell {
                background      = "color",
                backgroundcolor = (previous == current) and "colorone" or "colortwo",
                foregroundstyle = "bold",
                foregroundcolor = "white",
            }
            lxml.first(b,"/c")
            context.stopxcell()
            context.startxcell()
            lxml.first(b,"/d")
            context.stopxcell()
            previous = current
        context.stopxrow()
    end
    context.stopxtable()
end
\stopluacode

\startxmlsetups xml:test_setups
    \xmlsetsetup{#1}{a|b|c|d|i}{xml:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-5}{xml:test_setups}

\xmlprocessbuffer{example-5}{demo}{}
```

#1	Text
#2	More text
#2	Even more text
#2	And more
#3	And even more
#2	Something *nested*

The question is, do we really need LUA? Often we don't, apart maybe from an occasional special finalizer. A pure TEX solution is given next:

```
\startxmlsetups xml:a
    \glet\MyPreviousTitle\empty
    \glet\MyCurrentTitle \empty
    \startembeddedxtable
        \xmlflush{#1}
    \stopembeddedxtable
\stopxmlsetups

\startxmlsetups xml:b
    \startxrow
        \xmlflush{#1}
    \stopxrow
\stopxmlsetups

\startxmlsetups xml:c
    \xdef\MyCurrentTitle{\xmltext{#1}{.}}
    \doifelse {\MyPreviousTitle} {\MyCurrentTitle} {
        \startxcell
          [background=color,
           backgroundcolor=colorone,
           foregroundstyle=bold,
           foregroundcolor=white]
    } {
        \glet\MyPreviousTitle\MyCurrentTitle
        \startxcell
          [background=color,
           backgroundcolor=colortwo,
           foregroundstyle=bold,
           foregroundcolor=white]
    }
    \xmlflush{#1}
    \stopxcell
\stopxmlsetups

\startxmlsetups xml:d
```

```
    \startxcell
        \xmlflush{#1}
    \stopxcell
\stopxmlsetups

\startxmlsetups xml:i
    {\em\xmlflush{#1}}
\stopxmlsetups

\startxmlsetups xml:test_setups
    \xmlsetsetup{#1}{*}{xml:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-5}{xml:test_setups}

\xmlprocessbuffer{example-5}{demo}{}
```

#1	Text
#2	More text
#2	Even more text
#2	And more
#3	And even more
#2	Something *nested*

You can even save a few lines of code:

```
\startxmlsetups xml:c
    \xdef\MyCurrentTitle{\xmltext{#1}{.}}
    \startxcell
      [background=color,
       backgroundcolor=color\ifx\MyPreviousTitle\MyCurrentTitle one\else two\fi,
       foregroundstyle=bold,
       foregroundcolor=white]
    \xmlflush{#1}
    \stopxcell
    \glet\MyPreviousTitle\MyCurrentTitle
\stopxmlsetups
```

Or if you prefer:

```
\startxmlsetups xml:c
    \xdef\MyCurrentTitle{\xmltext{#1}{.}}
    \doifelse {\MyPreviousTitle} {\MyCurrentTitle} {
        \xmlsetup{#1}{xml:c:one}
    } {
```

```
        \xmlsetup{#1}{xml:c:two}
    }
\stopxmlsetups

\startxmlsetups xml:c:one
    \startxcell
      [background=color,
       backgroundcolor=colorone,
       foregroundstyle=bold,
       foregroundcolor=white]
    \xmlflush{#1}
    \stopxcell
\stopxmlsetups

\startxmlsetups xml:c:two
    \startxcell
      [background=color,
       backgroundcolor=colortwo,
       foregroundstyle=bold,
       foregroundcolor=white]
    \xmlflush{#1}
    \stopxcell
    \global\let\MyPreviousTitle\MyCurrentTitle
\stopxmlsetups
```

These examples demonstrate that it doesn't hurt to know a little bit of TEX programming: defining macros and basic comparisons can come in handy. There are examples in the test suite, you can peek in the source code, you can consult the wiki or you can just ask on the list.

<< 7.7 >> last match

For the next example we use the following XML input:

```
<?xml version "1.0"?>
<document>
    <section id="1">
        <content>
            <p>first</p>
            <p>second</p>
        </content>
    </section>
    <section id="2">
        <content>
            <p>third</p>
            <p>fourth</p>
        </content>
```

```
    </section>
</document>
```

If you check if some element is present and then act accordingly, you can end up with doing the same lookup twice. Although it might sound inefficient, in practice it's often not measureable.

```
\startxmlsetups xml:demo:document
    \type{\xmlall{#1}{/section[@id='2']/content/p}}\par
    \xmldoif{#1}{/section[@id='2']/content/p} {
        \xmlall{#1}{/section[@id='2']/content/p}
    }
    \type{\xmllastmatch}\par
    \xmldoif{#1}{/section[@id='2']/content/p} {
        \xmllastmatch
    }
    \type{\xmlall{#1}{last-match::}}\par
    \xmldoif{#1}{/section[@id='2']/content/p} {
        \xmlall{#1}{last-match::}
    }
    \type{\xmlfilter{#1}{last-match::/command(xml:demo:p)}}\par
    \xmldoif{#1}{/section[@id='2']/content/p} {
        \xmlfilter{#1}{last-match::/command(xml:demo:p)}
    }
\stopxmlsetups

\startxmlsetups xml:demo:p
    \quad\xmlflush{#1}\endgraf
\stopxmlsetups

\startxmlsetups xml:demo:base
    \xmlsetsetup{#1}{document|p}{xml:demo:*}
\stopxmlsetups

\xmlregisterdocumentsetup{example-6}{xml:demo:base}

\xmlprocessbuffer{example-6}{demo}{}
```

In the second check we just flush the last match, so effective we do an `\xmlall` here. The third and fourth alternatives demonstrate how we can use `last-match` as axis. The gain is 10% or more on the lookup but of course typesetting often takes relatively more time than the lookup.

```
\xmlall{example-6::3}{/section[@id='2']/content/p}
  third
  fourth
\xmllastmatch
  third
  fourth
```

```
\xmlall{example-6::3}{last-match::}
```
third
fourth
```
\xmlfilter{example-6::3}{last-match::/command(xml:demo:p)}
```
third
fourth

<< 7.8 >> Finalizers

The XML parser is also available outside TEX. Here is an example of its usage. We pipe the result to TEX but you can do with `t` whatever you like.

```
local x = xml.load("manual-demo-1.xml")
local t = { }

for c in xml.collected(x,"//*") do
    if not c.special and not t[c.tg] then
        t[c.tg] = true
    end
end

context.tocontext(table.sortedkeys(t))
```

This returns:

```
t={
 "content",
 "document",
 "p",
 "section",
 "title",
}
```

We can wrap this in a finalizer:

```
xml.finalizers.taglist = function(collected)
    local t = { }
    for i=1,#collected do
        local c = collected[i]
        if not c.special then
            local tg = c.tg
            if tg and not t[tg] then
                t[tg] = true
            end
        end
    end
    return table.sortedkeys(t)
```

```
end
```

Or in a more extensive one:

```
xml.finalizers.taglist = function(collected,parenttoo)
    local t = { }
    for i=1,#collected do
        local c = collected[i]
        if not c.special then
            local tg = c.tg
            if tg and not t[tg] then
                t[tg] = true
            end
            if parenttoo then
                local p = c.__p__
                if p and not p.special then
                    local tg = p.tg .. ":" .. tg
                    if tg and not t[tg] then
                        t[tg] = true
                    end
                end
            end
        end
    end
    return table.sortedkeys(t)
end
```

Usage is as follows:

```
local x = xml.load("manual-demo-1.xml")
local t = xml.applylpath(x,"//*/taglist()")

context.tocontext(t)
```

And indeed we get:

```
t={
 "content",
 "document",
 "p",
 "section",
 "title",
}
```

But we can also say:

```
local x = xml.load("manual-demo-1.xml")
local t = xml.applylpath(x,"//*/taglist(true)")
```

```
context.tocontext(t)
```

Now we get:

```
t={
 "content",
 "content:p",
 "document",
 "document:section",
 "p",
 "section",
 "section:content",
 "section:title",
 "title",
}
```