Dealing with xml in ConTEXt MkIV

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. We expect that the old method will be used less and less and eventually it might become a module in MkIV.

The user interface is sort of experimental but most commands discussed here are in use already in styles that we make and therefore these commands will stay. Over time we will add more examples to this document.

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 a mixup.

You may wonder why we do these manipulations in TEX and not use xslt instead. The advantage of an integrated approach is that it simplifies usage. Think of not only processing the a document, but also using xml for managing resources in the same run. Also, 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.

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.

Hans Hagen, Pragma ADE, August 2008

This is the first version of this manual. Some details of the implementation might change and this manual may contain errors.

Setting up a converter

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 non standard titles and formatting.

```
<?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>
```

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

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

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

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

\startxmlsetups xml:demo:section
  \section{\xmlfirst{#1}{/title}}
  \xmlfirst{#1}{/content}
\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.

For the moment stop wondering what some (empty) arguments are doing here. 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.

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:

```
\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 makes sense, especially when the `item` tag is used for other purposes as well.

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.

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][figures-here][figure]
\definefloat[figure-left][figures-left][figure]
\definefloat[table-here] [tables-here] [table]
\definefloat[table-left] [tables-left] [table]

\setupfloat[figure-here][default=here]
\setupfloat[figure-left][default=left]
\setupfloat[table-here] [default=here]
\setupfloat[table-left] [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.

Filtering content

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
  \xmlsetsetup{demo}{*}{-}
  \xmlfilter{demo}{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

\xmlregistersetup{first:demo:one}

\xmlprocessfile{demo}{music-collection.xml}{}
```

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's Money That Matters</track>
       <track length="156">Red Bandana</track>
      </tracks>
```

An alternative written in Lua looks as follows:

```
\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 r, d, k = xml.filter(m,p)
```

1. The xml file contains the collection stored on my slimserver instance.

```
  lxml.displayverbatim(d)
\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'm Dead (But I Don'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'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.

a few details

In ConTEXt 'setups' are a rather common variant on macros. An example of a setup is:

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

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

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 definition, we use a nicely formatted setup. The advantage of a setup is that spaces are ignored.

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

Commands

nodes and lpaths

The amount of commands available for manipulating the xml file is rather large. Many of the commands cooperate with so called 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: a document `id` (string) or an argument to a setup (result from a lookup). An `lpath` is a fancy name for a path expression (as with xslt) but resolved by Lua. A `filter` is an action that is applied to the result of a lookup.

loading

`\xmlload {id} {filename}` loads the file `filename` and registers it under `id`

`\xmlloadbuffer {id} {buffer}` loads the buffer `buffer` and registers it under `id`

`\xmlloaddata {id} {string}` loads `string` and registers it under `id`

`\xmlinclude {node} {lpath} {attribute}` includes the file specified by `attribute` of the element located by `lpath` at node `node`

`\xmlprocessfile {id} {filename} {initial-xml-setup}` registers file `filename` as `id` and process the tree starting with `initial-xml-setup`

`\xmlprocessbuffer {id} {buffer} {initial-xml-setup}` registers buffer `buffer` as `id` and process the tree starting with `initial-xml-setup`

`\xmlprocessdata {id} {string} {initial-xml-setup}` registers `string` as `id` and process the tree starting with `initial-xml-setup`

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).

`\xmlmain {id}` returns the whole documents

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

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.

`\xmlflush {node}` returns all nodes under `node`

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

`\xmltext {node} {lpath}` returns the text of the matching `lpath` under `node`

`\xmlall {node} {lpath}` returns all nodes under `node` that matches `lpath`

`\xmlfirst {node} {lpath}` returns the first node under `node` that matches `lpath`

`\xmllast {node} {lpath}` returns the last node under `node` that matches `lpath`

`\xmlfilter {node} {lpath/filter}` at a match of `lpath` a filter `filter` is applied and the result is returned

`\xmlsnippet {node} {n}` returns the n^{th} element under `node`

`\xmlindex {node} {lpath} {n}` returns the n^{th} match of `lpath` at node `node`; a negative number starts at the end

`\xmlconcat {node} {lpath} {text}` returns the sequence of nodes that match `lpath` at `node` whereby `text` is put between each match

`\xmlconcatrange {node} {lpath} {text} {n} {m}` returns the n^{th} upto m^{th} of nodes that match `lpath` at `node` whereby `text` is put between each match

`\xmlcommand {node} {lpath} {xml-setup-id}` apply the given setup to each match of `lpath` at node `node`

`\xmlstrip {node} {lpath}` remove leading and trailing spaces from nodes under `node` that match `lpath`

`\xmlstripped {node} {lpath}` remove leading and trailing spaces from nodes under `node` that match `lpath` and return the content afterwards

`\xmlstripnolines {node} {lpath}` remove leading and trailing spaces as well as collapse embedded spaces from nodes under `node` that match `lpath`

`\xmlstrippednolines {node} {lpath}` remove leading and trailing spaces as well as collapse embedded spaces from nodes under `node` that match `lpath` and return the content afterwards

`\xmlinlineverbatim {node} {lpath}` return the content of the `lpath` match as inline verbatim code, that is no further interpretation (expansion) takes place and spaces are honoured

`\xmldisplayverbatim {node} {lpath}` return the content of the `lpath` match as display verbatim code, that is no further interpretation (expansion) takes place and leading and trailing spaces and newlines are treated special

information

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

`\xmlname {node}` returns the complete name (including namespace prefix) of the given `node`

`\xmlnamespace {node}` returns the namespace of the given `node`

`\xmltag {node}` returns the tag of the element, without namespace prefix

`\xmltags {node} {lpath}` returns a comma-separated list of tags of elements that match the `lpath`

`\xmlcount {node} {lpath}` returns the number of matches of `lpath` at node `node`

`\xmlnofelements {node}` returns the number of elements at node `node`

`\xmlatt {node} {name}` returns the value of attribute `name` or empty if no such attribute exists

`\xmlattdef {node} {name} {default}` returns the value of attribute `name` or `default` if no such attribute exists

`\xmlattribute {node} {lpath} {name}` finds a first match for `lpath` at `node` and returns the value of attribute `name` or empty if no such attribute exists

`\xmlattributedef {node} {lpath} {name} {default}` finds a first match for `lpath` at `node` and returns the value of attribute `name` or `default` if no such attribute exists

manipulation

You can use Lua code to manipulate the tree and it makes no sense to duplicate this in TEX. So, we only provide an interface to the most useful manipulators.

`\xmldelete {node} {lpath}` deletes all children of `node` that match `lpath`

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.

`\xmlloaddirectives {filename}` loads ConTEXt directives from `filename` 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 `setup` you can specify `before` and `after`.

`\xmldirectives {node} {lpath}` apply the setups directive associated with the found nodes

`\xmldirectivesbefore {node} {lpath}` apply the before directives associated with the found nodes

`\xmldirectivesafter {node} {lpath}` apply the after directives associated with the found nodes

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. You can also invoke such directives directly:

`\xmlcontextdirective {kind} {class} {key} {value}` execute the directive associated with `kind` and pass three arguments to it

This assumes that there is a command `xmlkinddirective` or in the MathML example `xmlmathmldirective` that does something useful.

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 entity and not depend on it being a number. Just think of it as 'the current node'. You can (and probably will) call such setups directly:

`\xmlsetup {name} {node}` expands setup `name` 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.

`\xmlsetfunction {name} {lpath} {function}` associates function Lua `function` to the elements in namespace `name` that match `lpath`

`\xmlsetsetup {name} {lpath} {setup}` associates setups (TEX code) `setup` to the elements in namespace `name` that match `lpath`

`\xmlprependsetup {setup}` pushes `setup` to the front of global list of setups to be applied

`\xmlappendsetup {setup}` pushes `setup` to the end of global list of setups to be applied

`\xmlbeforesetup {setup} {position}` inserts `setup` before setup `position` in the global list of setups to be applied

`\xmlaftersetup {setup} {position}` inserts `setup` after setup `position` in the global list of setups to be applied

`\xmlremovesetup {setup}` removes `setup` from the global list of setups to be applied

`\xmlprependdocumentsetup {id} {setup}` pushes `setup` to the front of `id` specific list of setups to be applied

`\xmlappenddocumentsetup {id} {setup}` pushes `setup` to the end of `id` specific list of setups to be applied

`\xmlbeforedocumentsetup {id} {setup} {position}` inserts `setup` before setup `position` in the `id` specific list of setups to be applied

`\xmlafterdocumentsetup {id} {setup} {position}` inserts `setup` after setup `position` in the `id` specific list of setups to be applied

`\xmlremovedocumentsetup {setup}` removes `setup` from the `id` specific list of setups to be applied

`\xmlresetdocumentsetups {id}` removes all setups from the `id` specific list of setups to be applied

`\xmlflushdocumentsetups {id}` applies all setups in tagged with `id`

`\xmlregisteredsetups` applies all global setups to the current document

`\xmlregistereddocumentsetups` applies all document specific setups to the current document

testing

The following test macros all take a `node` as first argument and an `lpath` as second:

`\xmldoif {node} {lpath} {yes}` expands to `yes` when `lpath` matches at node `node`

`\xmldoifnot {node} {lpath} {no}` expands to `no` when `lpath` does not match at node `node`

`\xmldoifelse {node} {lpath} {yes} {no}` expands to `yes` when `lpath` matches at node `node` and to `no` otherwise

`\xmldoiftext {node} {lpath} {yes}` expands to `yes` when the node matching `lpath` at node `node` has some content

`\xmldoifnottext {node} {lpath} {no}` expands to `do-if-fase` when the node matching `lpath` at node `node` has no content

`\xmldoifelsetext {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` has content and to `no` otherwise

`\xmldoifelseempty {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` is empty and to `no` otherwise

`\xmldoifelseselfempty {node} {lpath} {yes} {no}` expands to `yes` when the node matching `lpath` at node `node` is empty and to `no` otherwise

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.

When you set `method` to `mkiv`, the traditional handler will not kick in when xml code ends up in TEX. When we have replaced all usage of the MkII method in the core of ConTEXt, we might make this default.

When you set `default` to `text` elements with no setup assigned will end up as text. When set to `none` such elements will be hidden. When no value is set the outcome depends on the method: interpreted as xml in for `mkii` and text for method `mkiv`.

You can set `compress` to `yes` in which case comment is stripped from the tree when the file is read. When `entities` is set to `yes` (this is the default) entities are replaced.

`\xmlregisterns {internal} {public}` associates an internal namespace (like `mml`) with one given in the document as url (like mathml)

`\xmlremapname {node} {lpath} {new-namespace} {new-tag}` changes the namespace and tag of the matching elements

`\xmlremapnamespace {node} {lpath} {from} {to}` replaces all references to the given namespace to a new one

`\xmlchecknamespace {id} {lpath} {new}` sets the namespace of the matching elements unless a namespace is already set

`\xmldefaulttotext {id}` makes all elements that don't have a setup associated resolve to text

`\xmldefaulttonone {id}` hides all elements that don't have a setup associated

`\xmlutfize {id}` convert all entities to utf if possible

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, for instance onto strings or commands.

`\xmlmapvalue {category} {name} {value}` associate a `value` with a `name` and `category`

`\xmlvalue {category} {name} {default}` expand the value `value` associated with a `category` and `name` and if not resolved, expand `default`

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.

synonyms

A few of the discussed commands have synonyms

`\xmlmapval`	`\xmlmapvalue`
`\xmlval`	`\xmlvalue`
`\xmlregistersetup`	`\xmlappendsetup`
`\xmlregisterdocumentsetup`	`\xmlappenddocumentsetup`

Expressions and filters

path expressions

In the previous sections we used `lpath` expressions, which are a variant on `xpath` expressions as in xslt but in this case more geared towards usage in TEX. These mechanisms will be extended when needed.

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:

`position`	number	the current index of the matched element
`index`	number	the current index upto the matched element
`text`	string	the textual representation of the matched element
`name`	string	the full name of the matched element: namespace and tag
`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

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

```
xml.expressions.contains = string.find
xml.expressions.find     = string.find
xml.expressions.upper    = string.upper
xml.expressions.lower    = string.lower
xml.expressions.number   = tonumber
xml.expressions.boolean  = toboolean -- mkiv specific
```

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.

r	table	the root of the element
d	table	the roots data table
k	number	the current index into the roots data table
e	table	the element (d[k])
ns	string	the namespace (e.rn or e.ns)
tg	string	the tag (e.tg)
dt	table	the content (e.dt)
at	table	a hash containing the attributes (e.at)
id	number	the current elements index (not counting text)
ps	number	the current elements position (differs from id if mixed elements)
tx	string	the (first) text (dt[1])

The given expression between [] is converted to a Lua expression so you can use the usual ingredients:

```
== ~= <= >= < > 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.

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 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 section.

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. Examples of functions are:

text	string	returns the content
name	string	returns the (either or not remapped) namespace
ns	string	returns gives the original namespace
tag	string	returns the elements name
count	number	returns the elements name

Not all such functions make sense in TEX, for instance because they return a data structure that is useless for TEX itself. Instead of using functions like first(), you can as well use \xmlfirst which might be more efficient.

attribute(name)	returns the attribute with the given name
command(name)	expands the setup with the given name for each found element
position(n)	processes the n^{th} instance of the found element
first()	processes the first instance of the found element
last()	processes the last instance of the found element

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.

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.

Hans Hagen
Pragma ADE, Hasselt