Hans Hagen, Pragma ADE, October 28, 2009

Dealing with XML in ConTeXt MkIV

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. 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. 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 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 mechanism described here is still somewhat experimental and will be cleaned up and improved. In the case of resolved bugs you might need to upgrade your styles accordingly.

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 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>
```

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{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
  \chapter{\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. The argument #1 represents the current node (element) and is unique.

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 comples itemize and figure placement.

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:

```
\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.

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
  \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>
```

[1] The xml file contains the collection stores on my slimserver instance.

```
  <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 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'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.

2.2 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 definitions, we use a nicely formatted setup. A nice feature of a setup is that spaces are ignored so you don't need to worry about spurious spaces.

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

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 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). A `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.

3.2 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.

3.3 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 valus 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

3.4 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

3.5 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.

3.6 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.

3.7 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

3.8 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

3.9 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 `default` to `text` elements with no setup assigned will end up as text. When set to `hidden` such elements will be hidden.

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

3.10 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.

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

`\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.

3.11 synonyms

A few of the discussed commands have synonyms

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

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`	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
`index`	number	the current index of the matched element in its parent list
`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

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:

```
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.

`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 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.

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. 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 the somewhat less efficient `\xmlfirst` and friends.

`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.

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 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:

```
\xmlshow{#1}
\xmlname{#1}
```

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.profile`	report all parsed lpath expressions (in the log)
`xml.profile`	report all parsed lpath expressions (in the log)
`xml.profile`	report all parsed lpath expressions (in the log)
`xml.profile`	report all parsed lpath expressions (in the log)
`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

6 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.

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}]
    \or
        % index is used for access via setup
        \setuphead[chapter][expansion=yes,xmlsetup=xml:title:flush]
        \startchapter[title=\xmlgetindex{#1}]
    \or
        % tex call to xml using index is used
        \setuphead[chapter][expansion=yes]
        \startchapter[title=hm: \xmlreference{#1}{xml:title:flush}]
    \fi
    \xmlfirst{#1}{content}
    \stopchapter
\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.

7 Example paths

There is not that much system in the following examples. They resulted from tests with different documents. The current implementation evolved out if 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.

```
pattern: /*
```

```
1  axis  child
```

```
pattern: /**
```

```
1  axis  descendant
```

```
pattern: answer
```

```
1  axis   auto-descendant-or-self
2  nodes  *:answer
```

```
pattern: answer/test/*
```

```
1  axis   auto-descendant-or-self
2  nodes  *:answer
```

```
3  axis   auto-child
4  nodes  *:test
5  axis   child
```

pattern: answer/test/child::

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

pattern: answer/*

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

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

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

```
pattern: **

1  axis  descendant
```

```
pattern: *

1  axis  child
```

```
pattern: ..

1  axis  parent
```

```
pattern: .

1  axis  self
```

```
pattern: //

1  axis  descendant-or-self
```

```
pattern: /
```

```
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
```

Example paths

```
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
```

Example paths

```
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
```

```
pattern: one//two

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

```
pattern: one/*/two

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

```
pattern: one/**/two

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

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

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

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

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

```
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

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

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

```
pattern: something/whatever
```

1	axis	auto-descendant-or-self
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-or-self
2	nodes	*:whoknows

```
3  axis   auto-child
4  nodes  *:whocares
```

pattern: whoknows

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

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

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

pattern: whocares/first()

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

pattern: whocares/last()

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

pattern: whatever/all()

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

pattern: whocares/position(2)

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

pattern: whocares/position(-2)

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

`pattern: whocares[1]`

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

`pattern: whocares[-1]`

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

`pattern: whocares[2]`

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

`pattern: whocares[-2]`

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

`pattern: whatever[3]/attribute(id)`

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

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

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

`pattern: whatever[3]/text()`

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

pattern: whatever/all()

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

Example paths

`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
```