Transforming TEI documents

Sebastian Rahtz

October 2005

Accessing your TEI document

So you've created some TEI XML documents, what now?

- XPath
- XML Query (XQuery)
- XSLT Tranformation to another format (HTML, PDF, RTF, CSV, etc.)
- Custom Applications (Xaira, TEIPubisher, Philologic etc.)

What is XPath?

- It is a syntax for accessing parts of an XML document
- It uses a path structure to define XML elements
- It has a library of standard functions
- It is a W3C Standard
- It is one of the main components of XQuery and XSLT

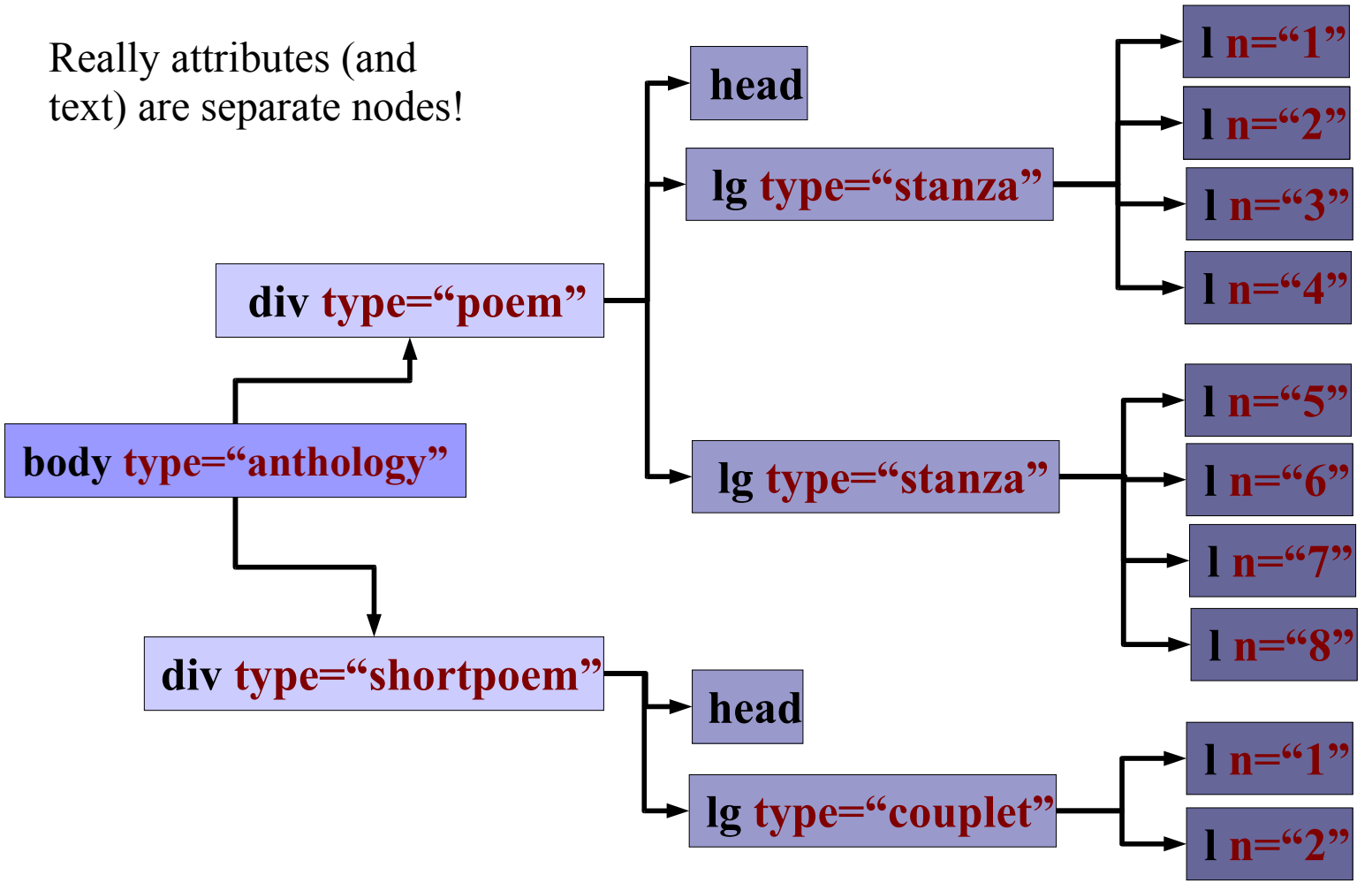
Example text


```
<body type="anthology">
 <div type="poem">
  <head>The SICK ROSE </head>
  <lg type="stanza">
   <l n="1">O Rose thou art sick.</l>
   <l n="2">The invisible worm,</l>
   <l n="3">That flies in the night </l>
   <l n="4">In the howling storm:</l>
  </lg>
  <lg type="stanza">
   <l n="5">Has found out thy bed </l>
   <l n="6">Of crimson joy:</l>
   <l n="7">And his dark secret love </l>
   <l n="8">Does thy life destroy.</l>
  </lg>
 </div>

</body>
```

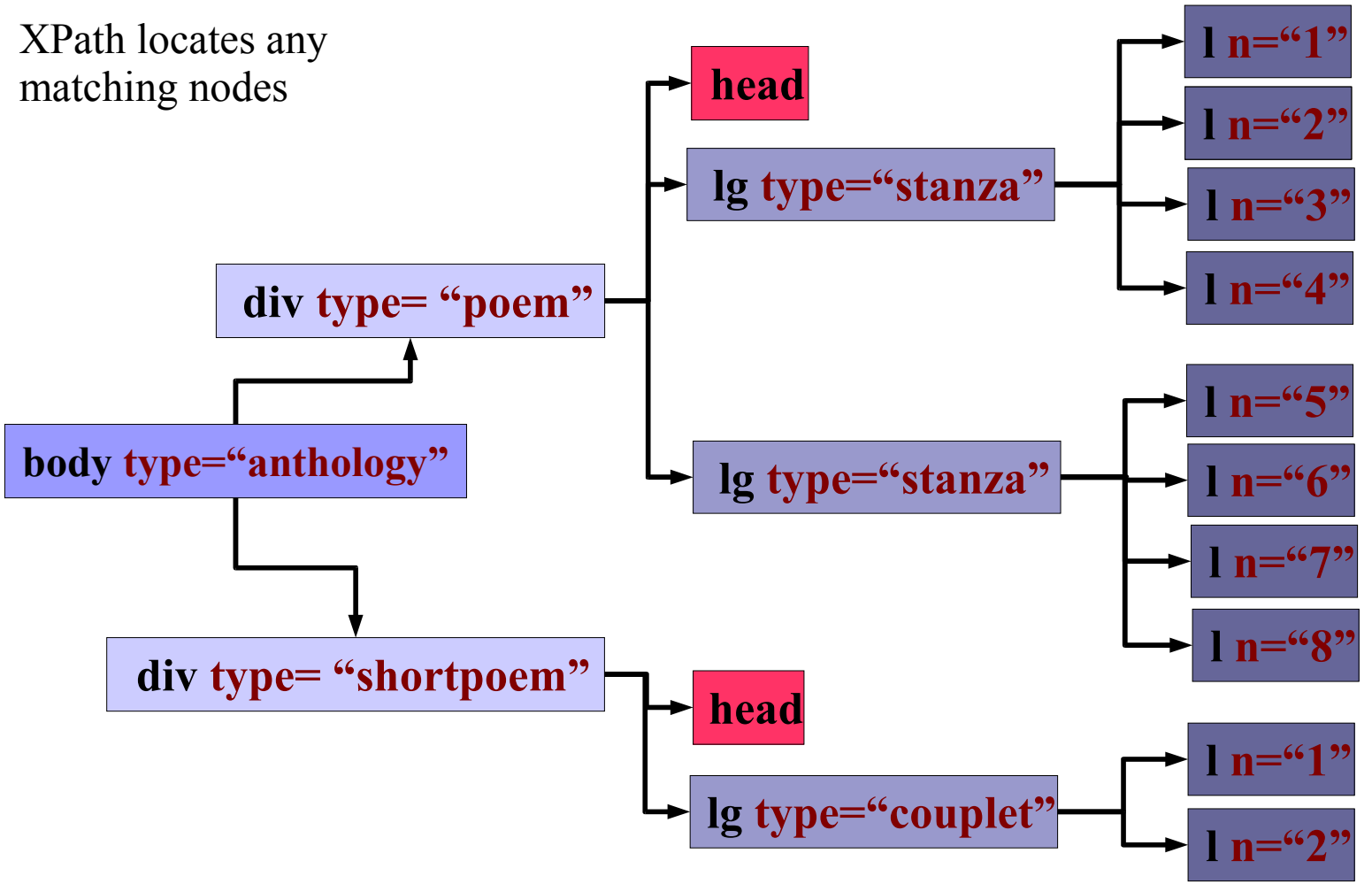
XML Structure

Really attributes (and text) are separate nodes!

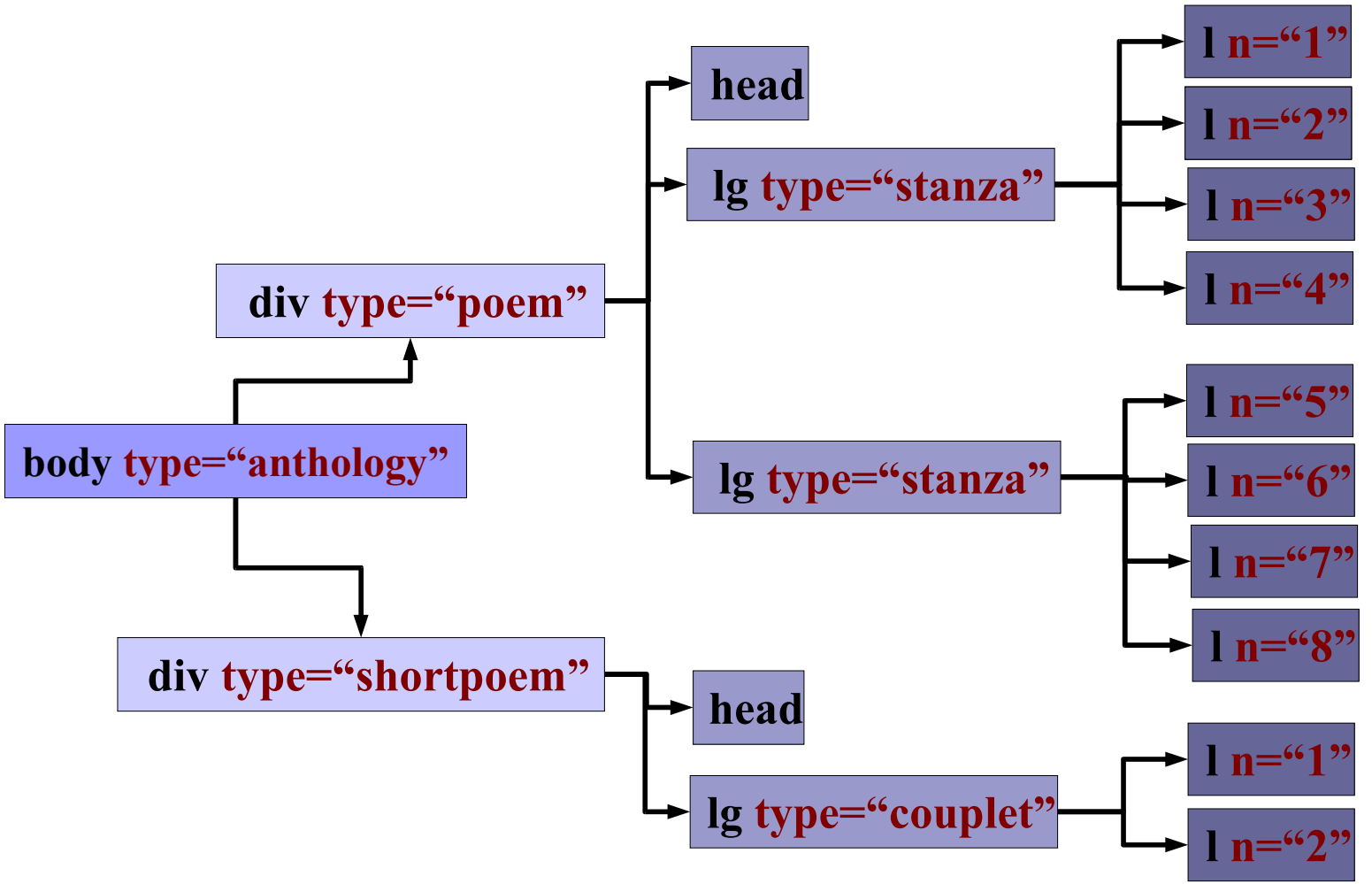


/body/div/head

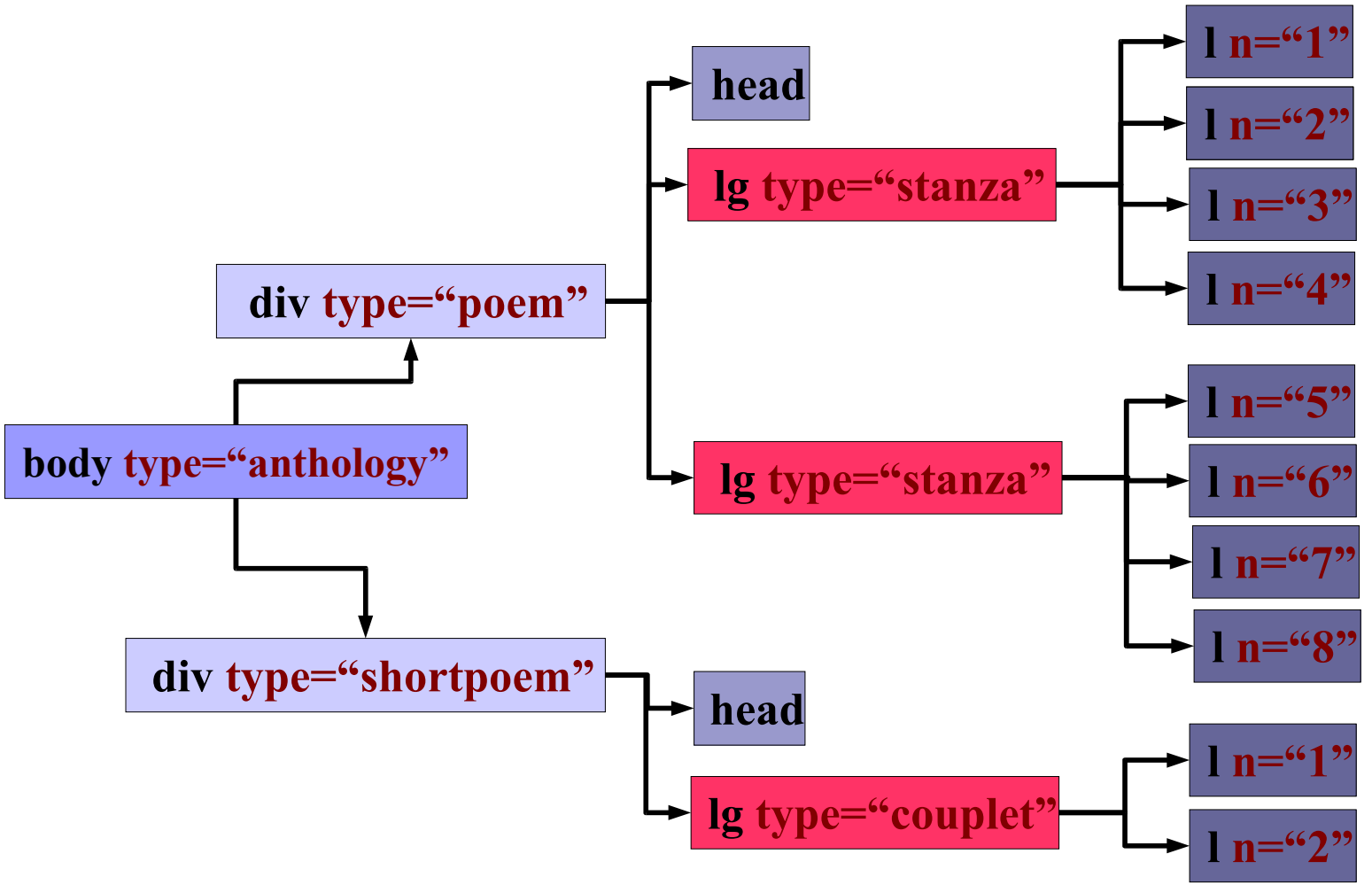
XPath locates any matching nodes



/body/div/lg ?

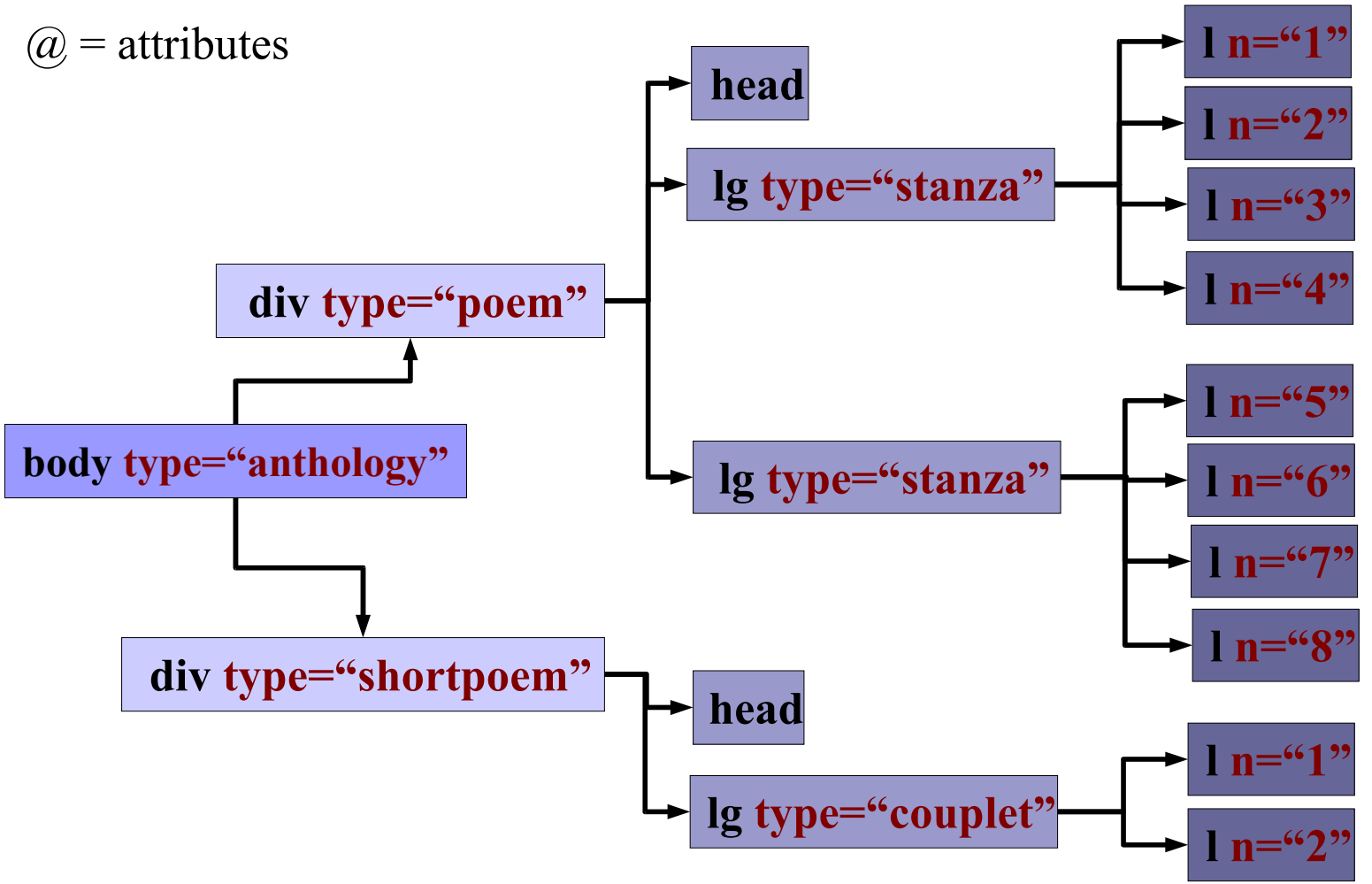

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

/body/div/lg

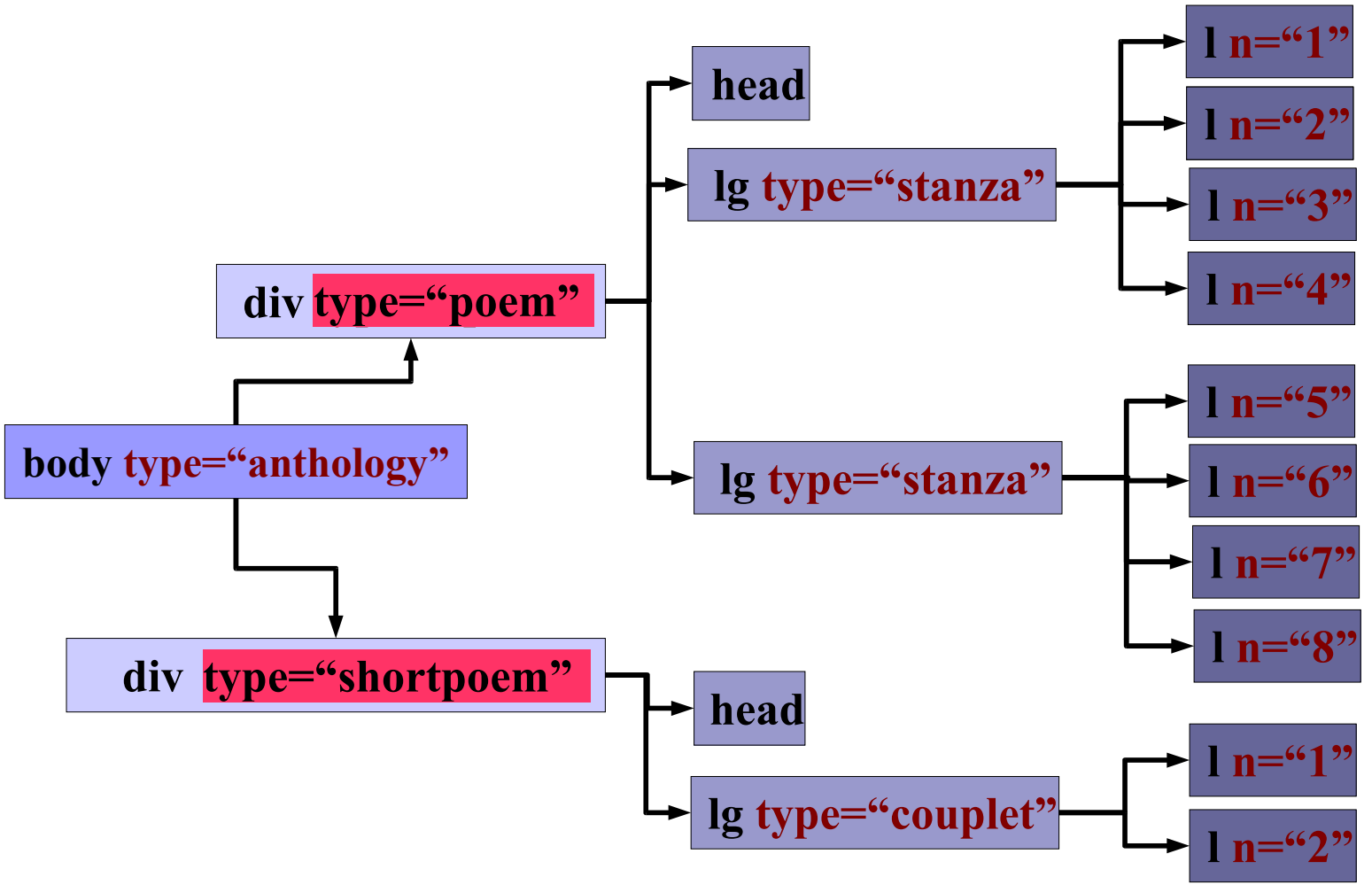

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

/body/div/@type ?

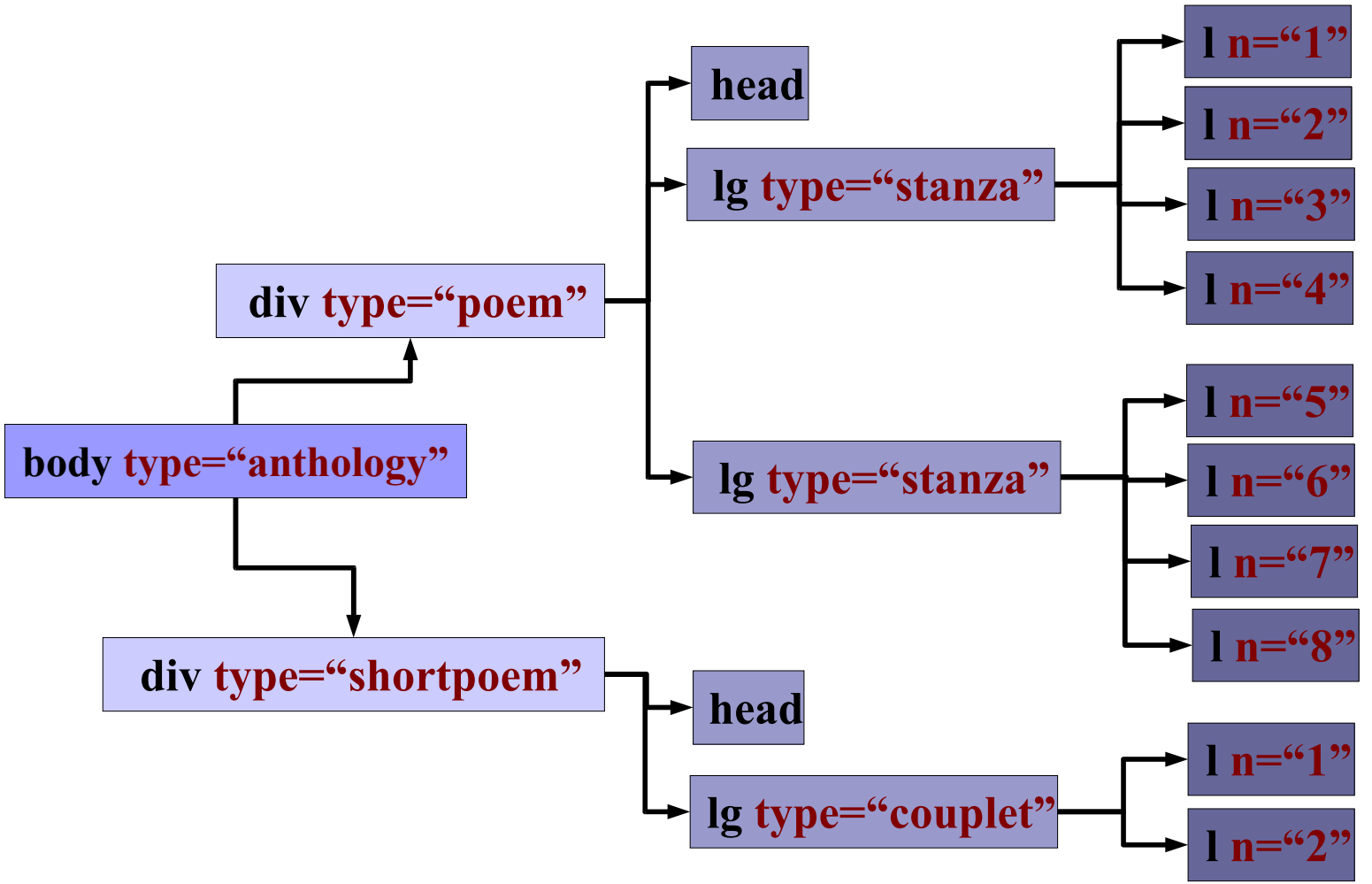
@ = attributes



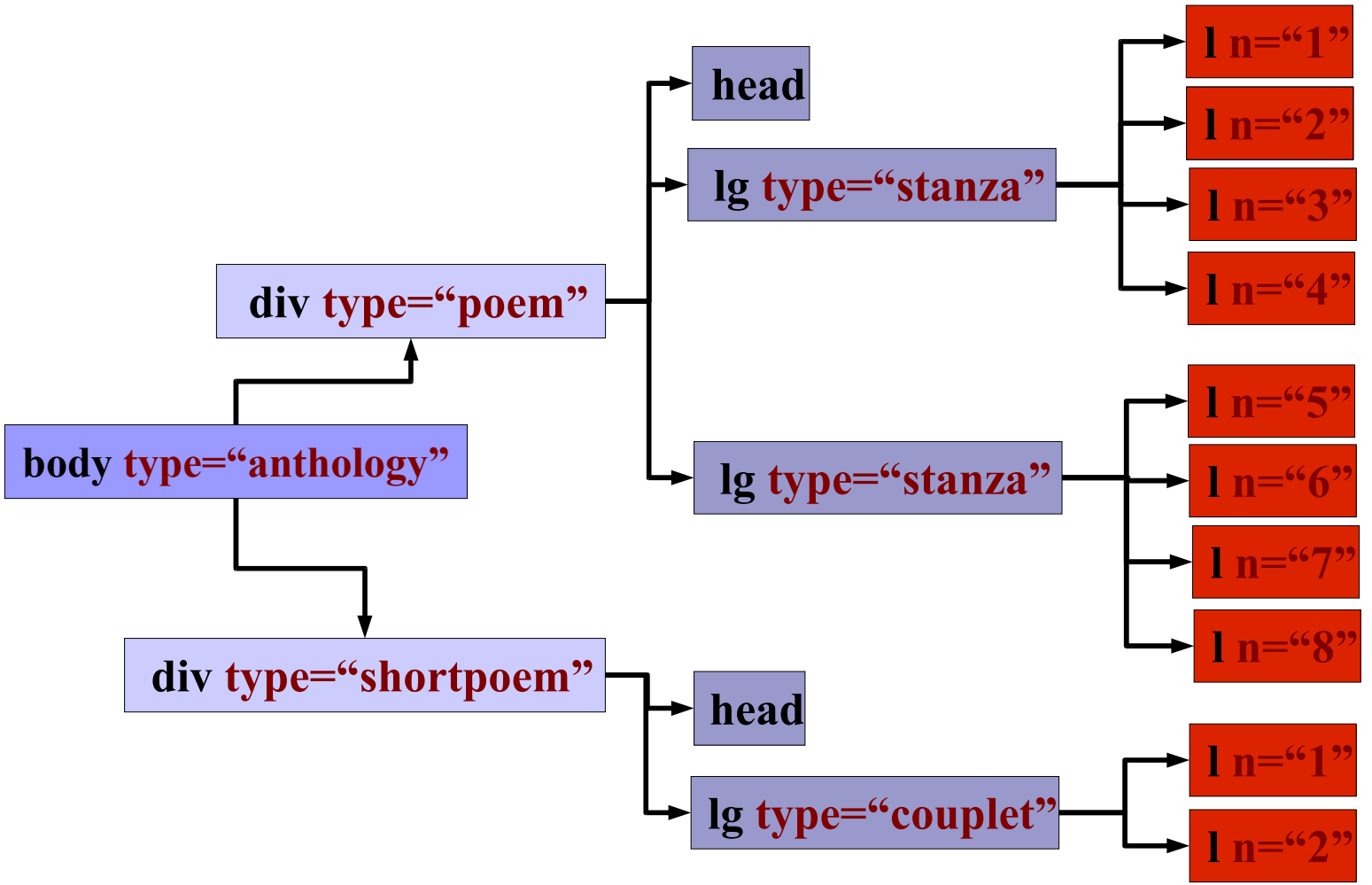
/body/div/@type



/body/div/lg/l ?

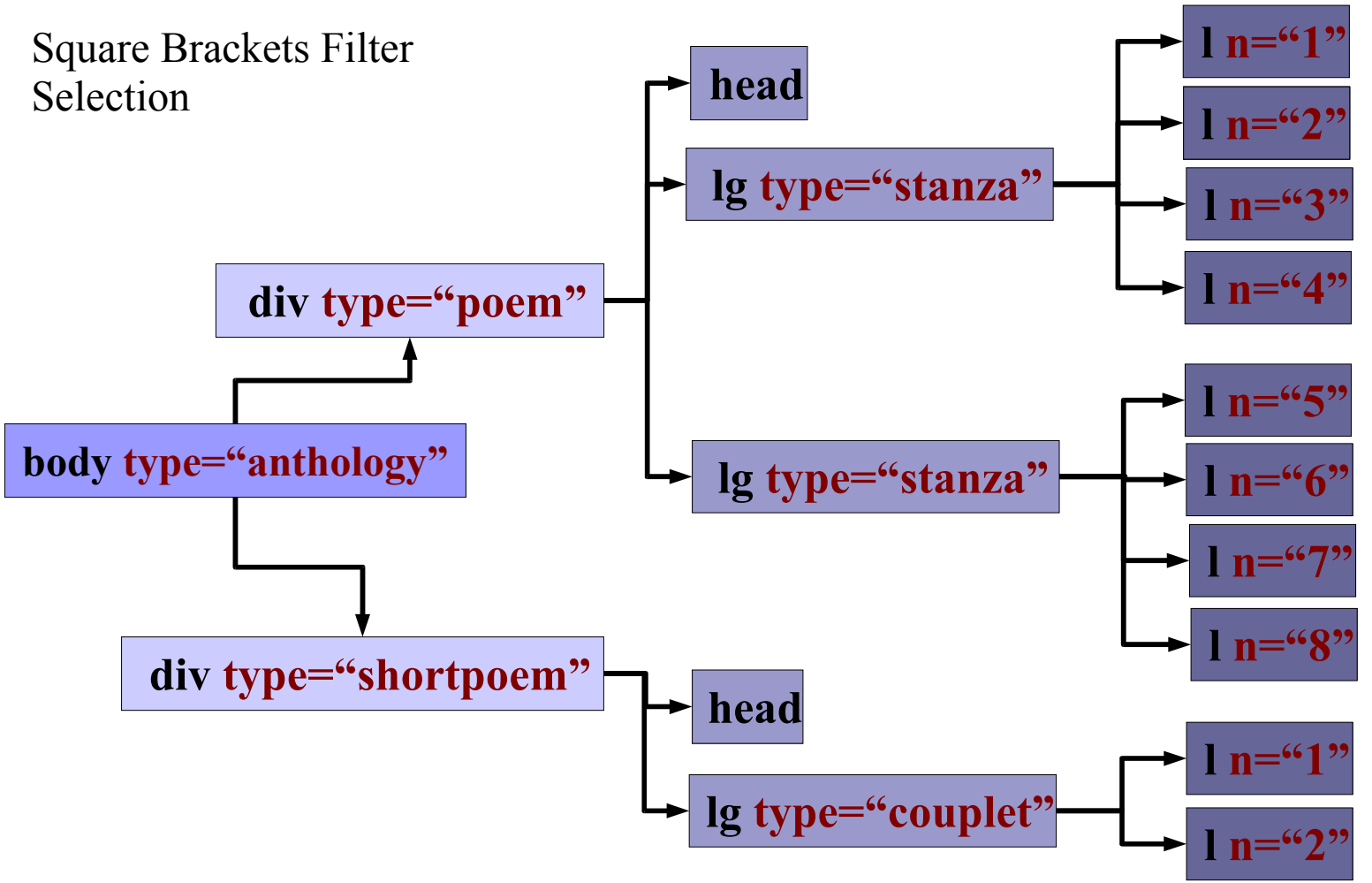

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

/body/div/lg/l

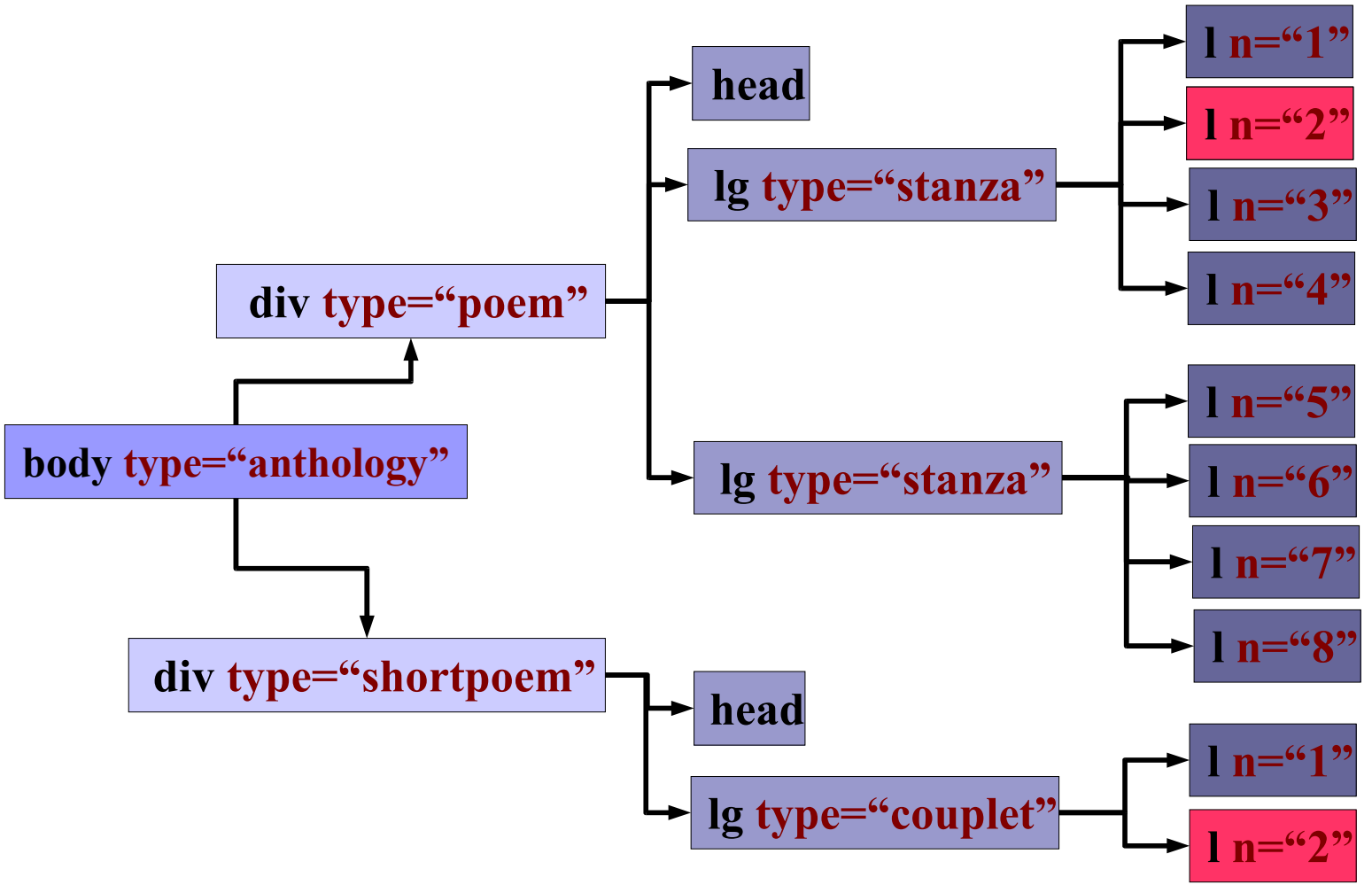

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

/body/div/lg/l[@n="2"] ?

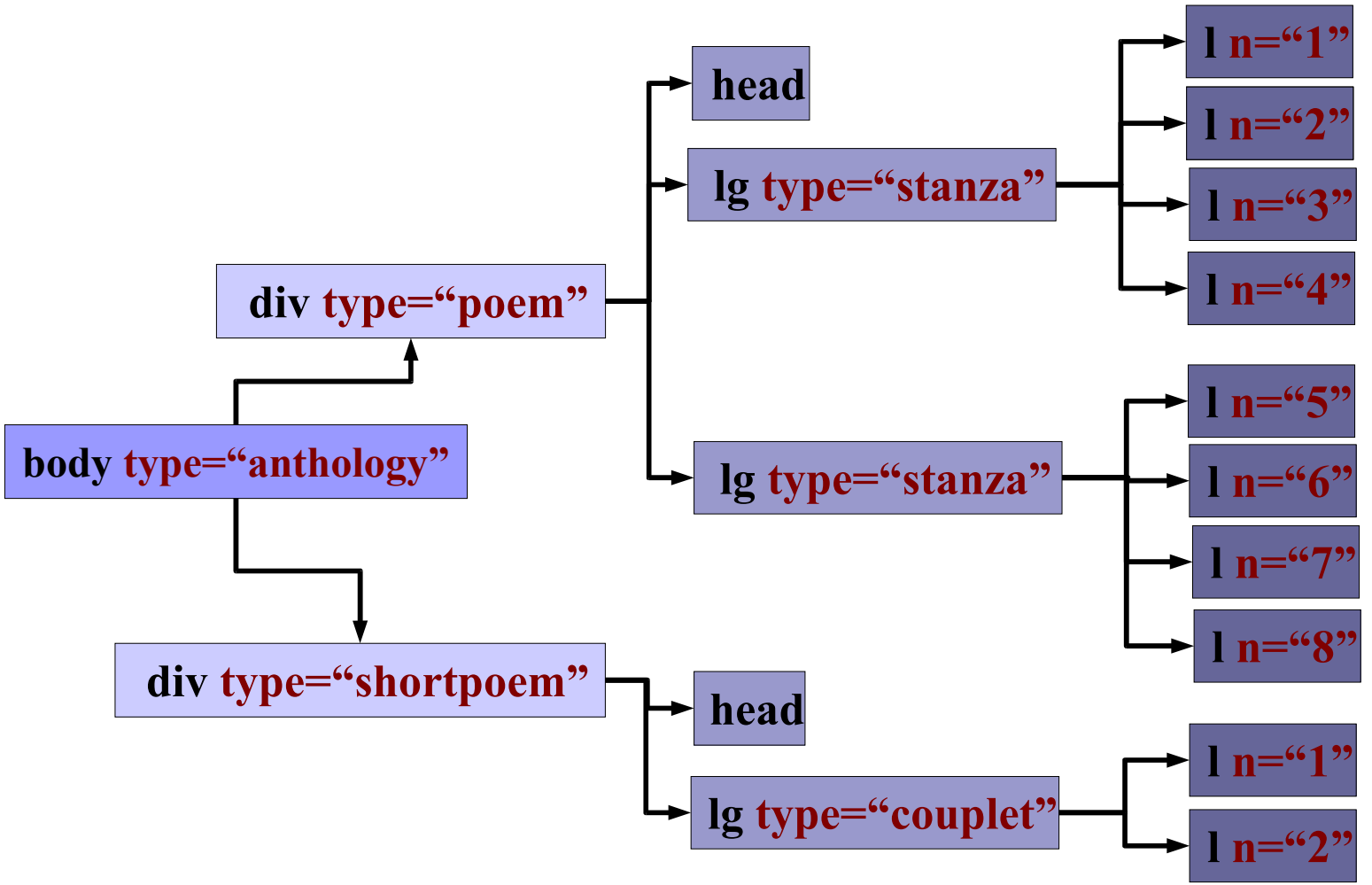
Square Brackets Filter Selection



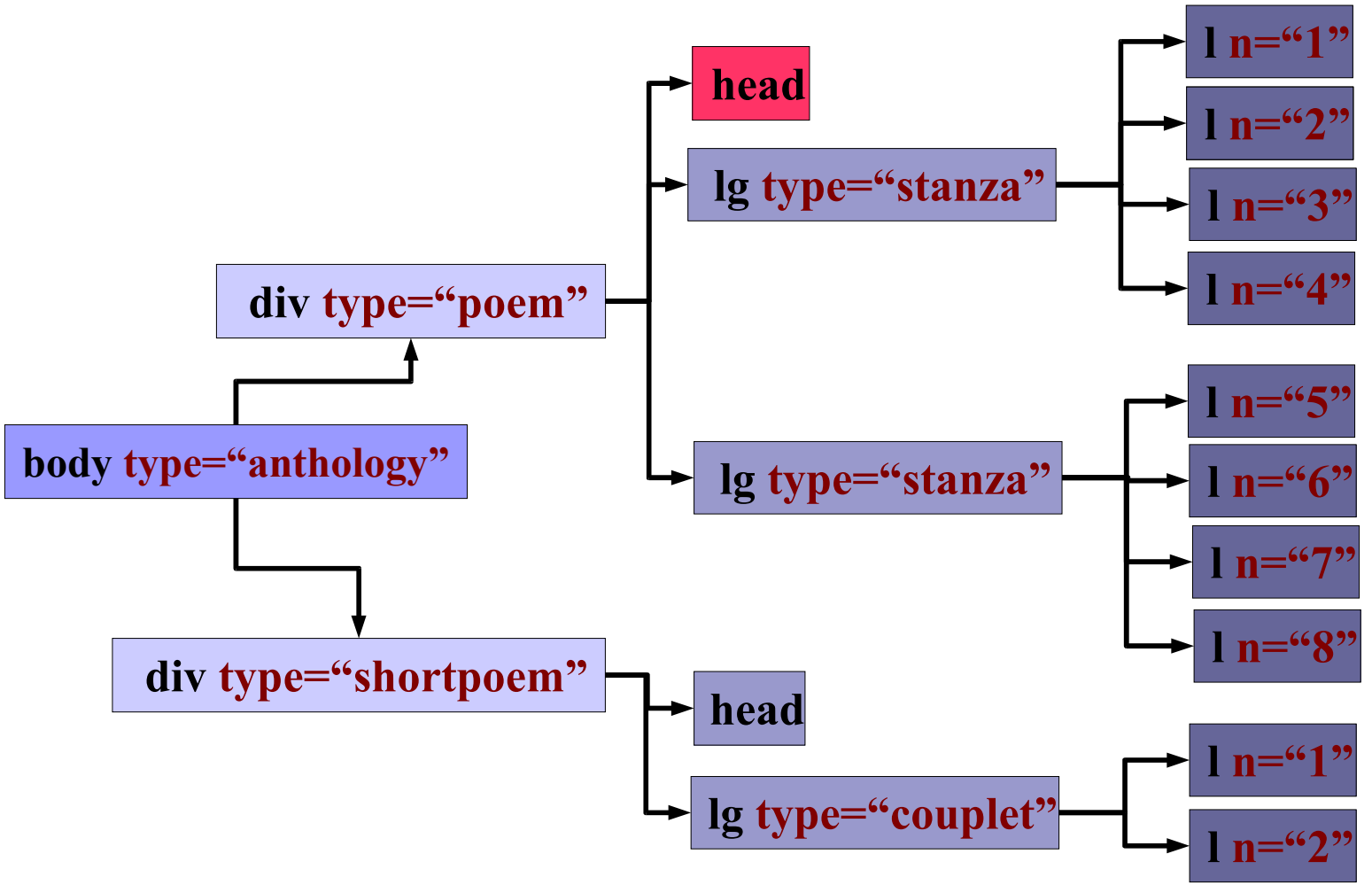
/body/div/lg/l[@n="2"]



/body/div[@type="poem"]/head ?

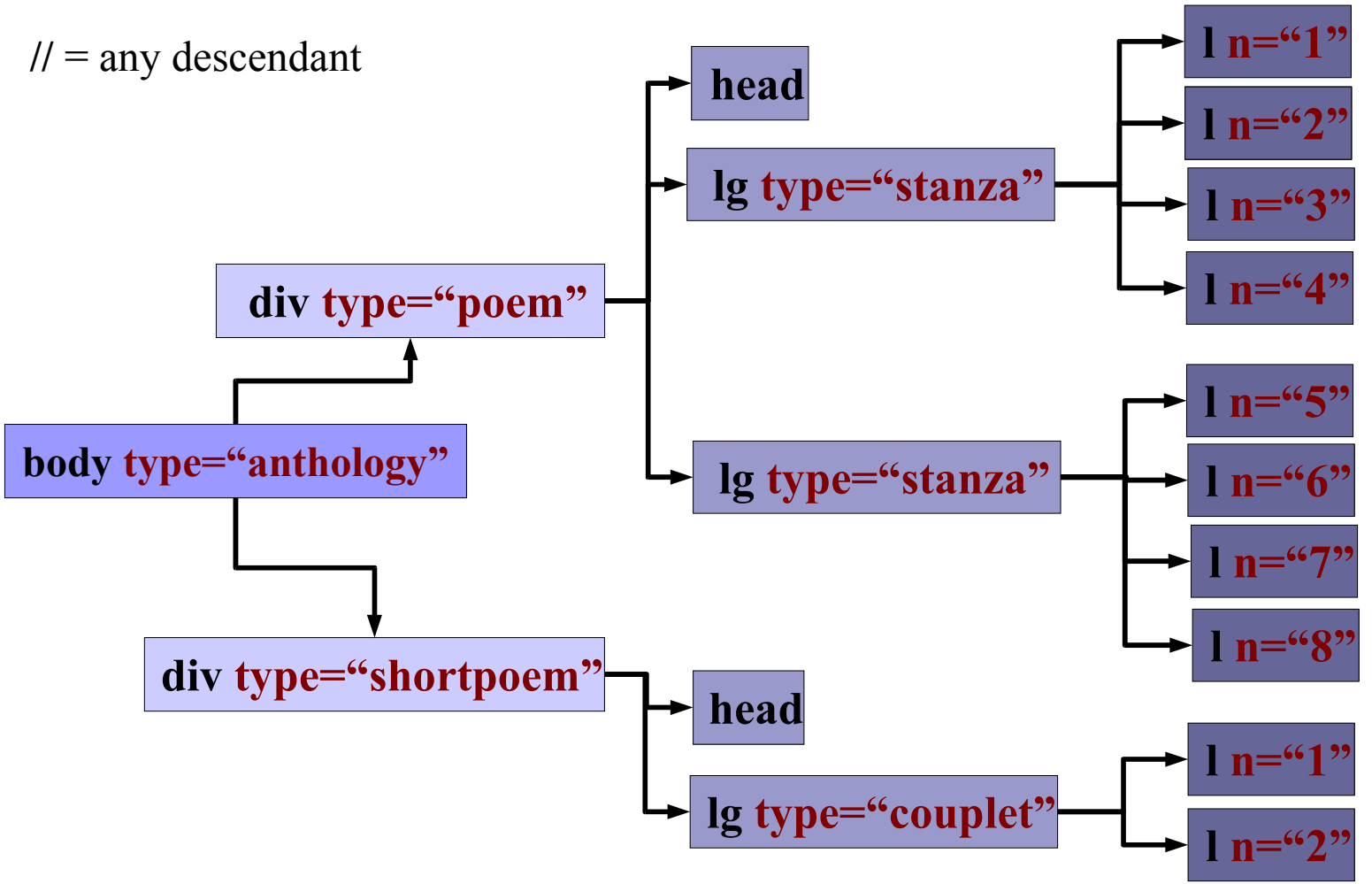


/body/div[@type="poem"]/head

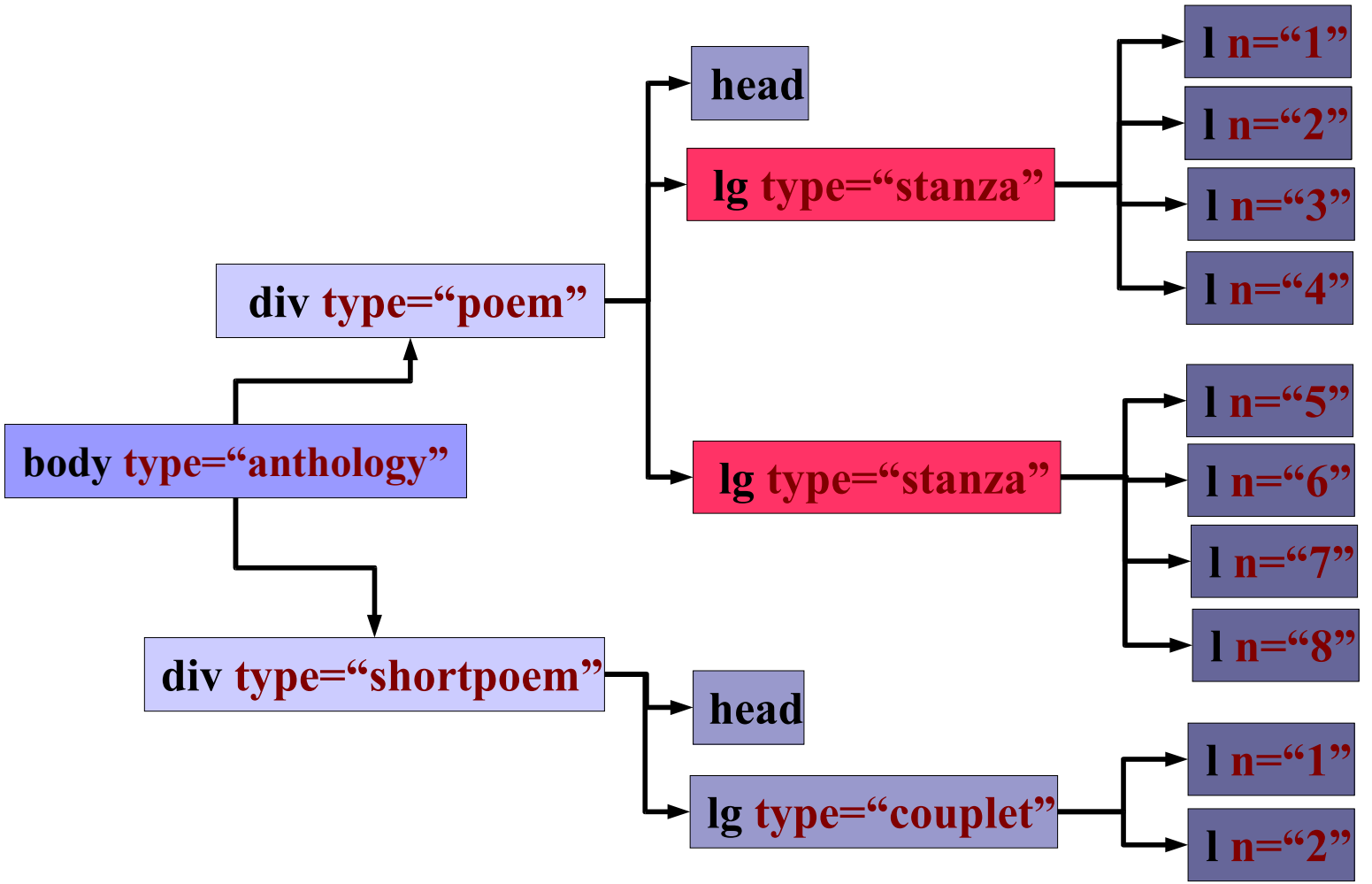


//lg[@type="stanza"] ?

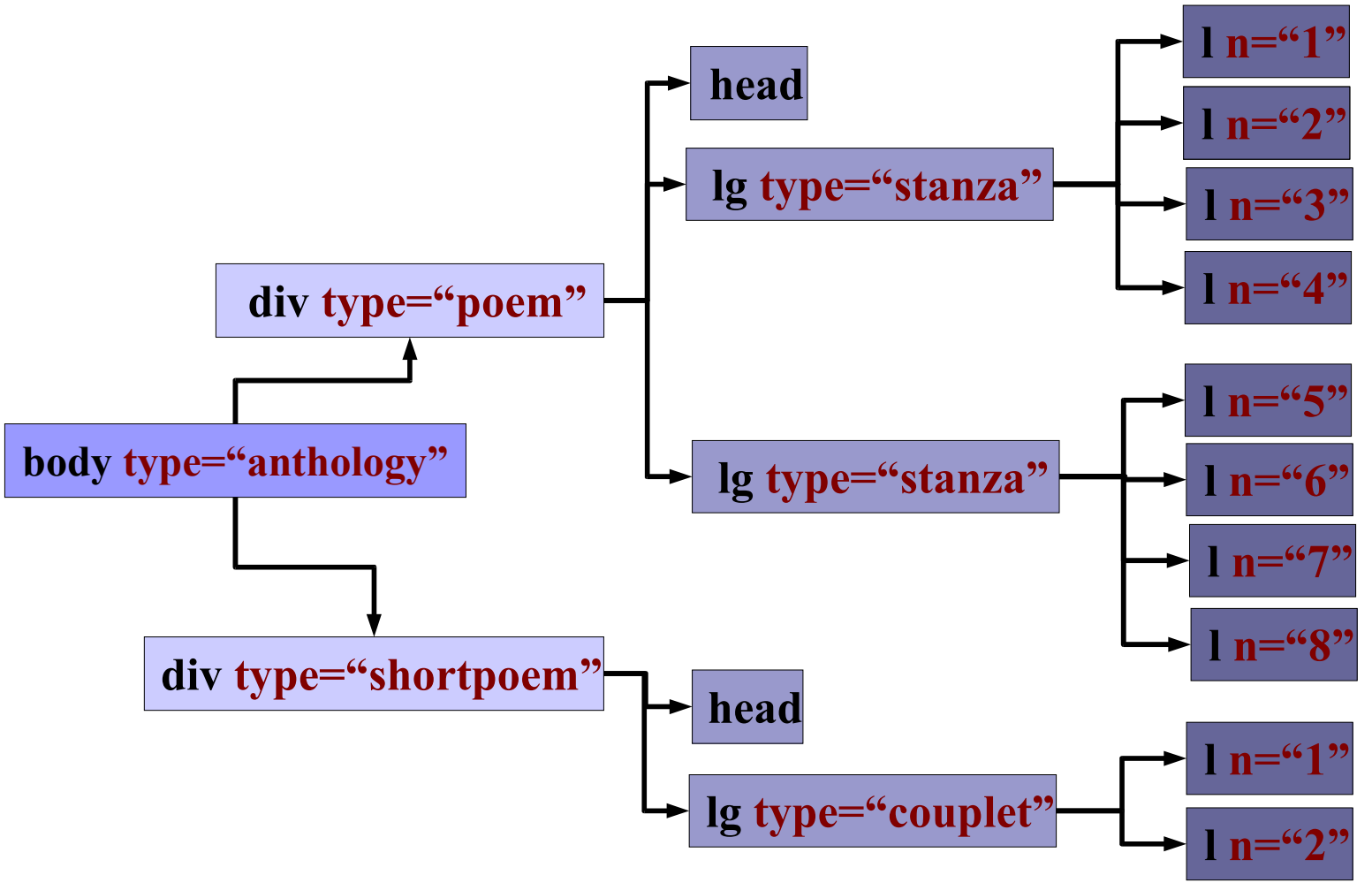
// = any descendant



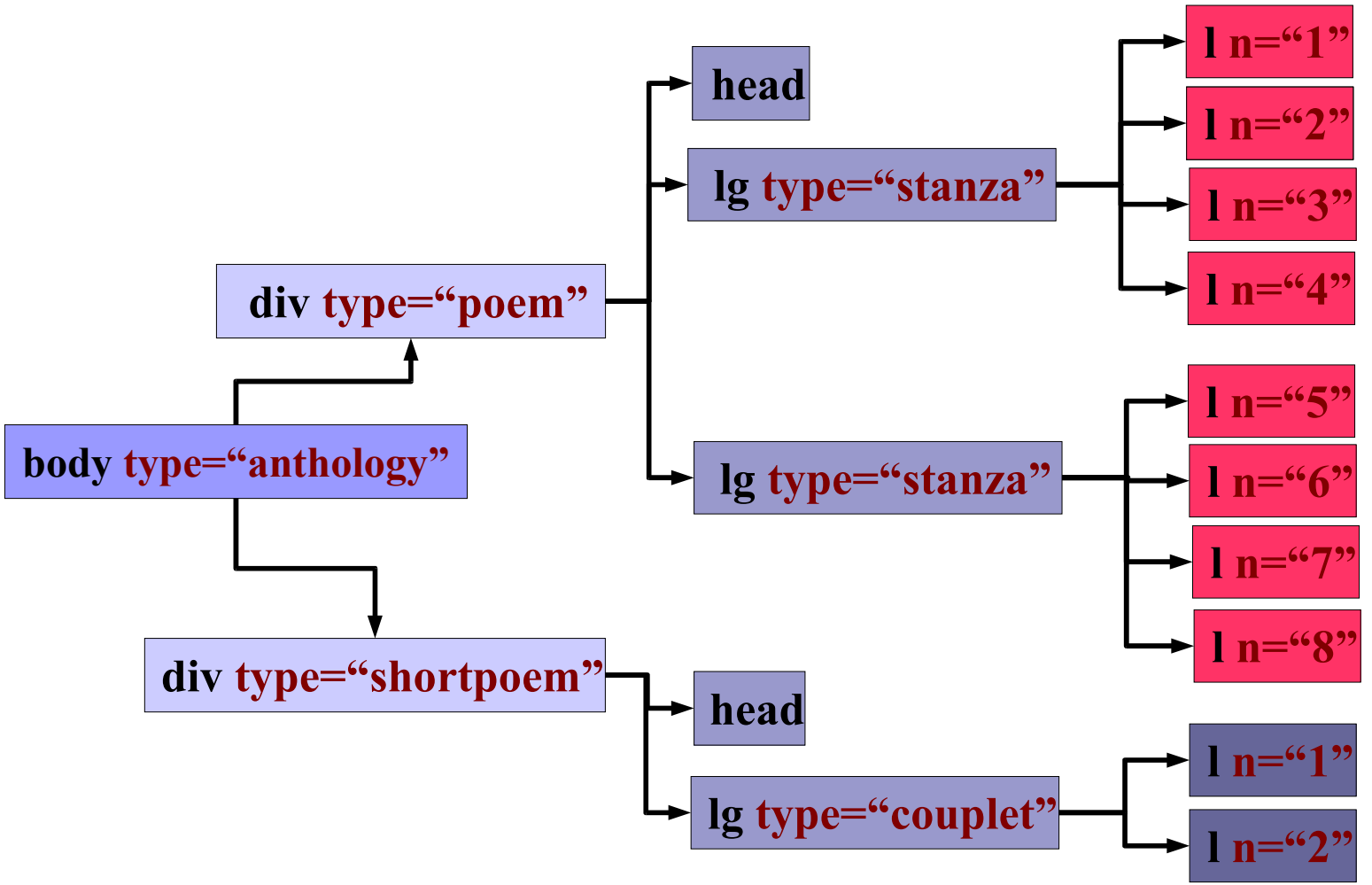
//lg[@type="stanza"]



//div[@type="poem"]//l ?

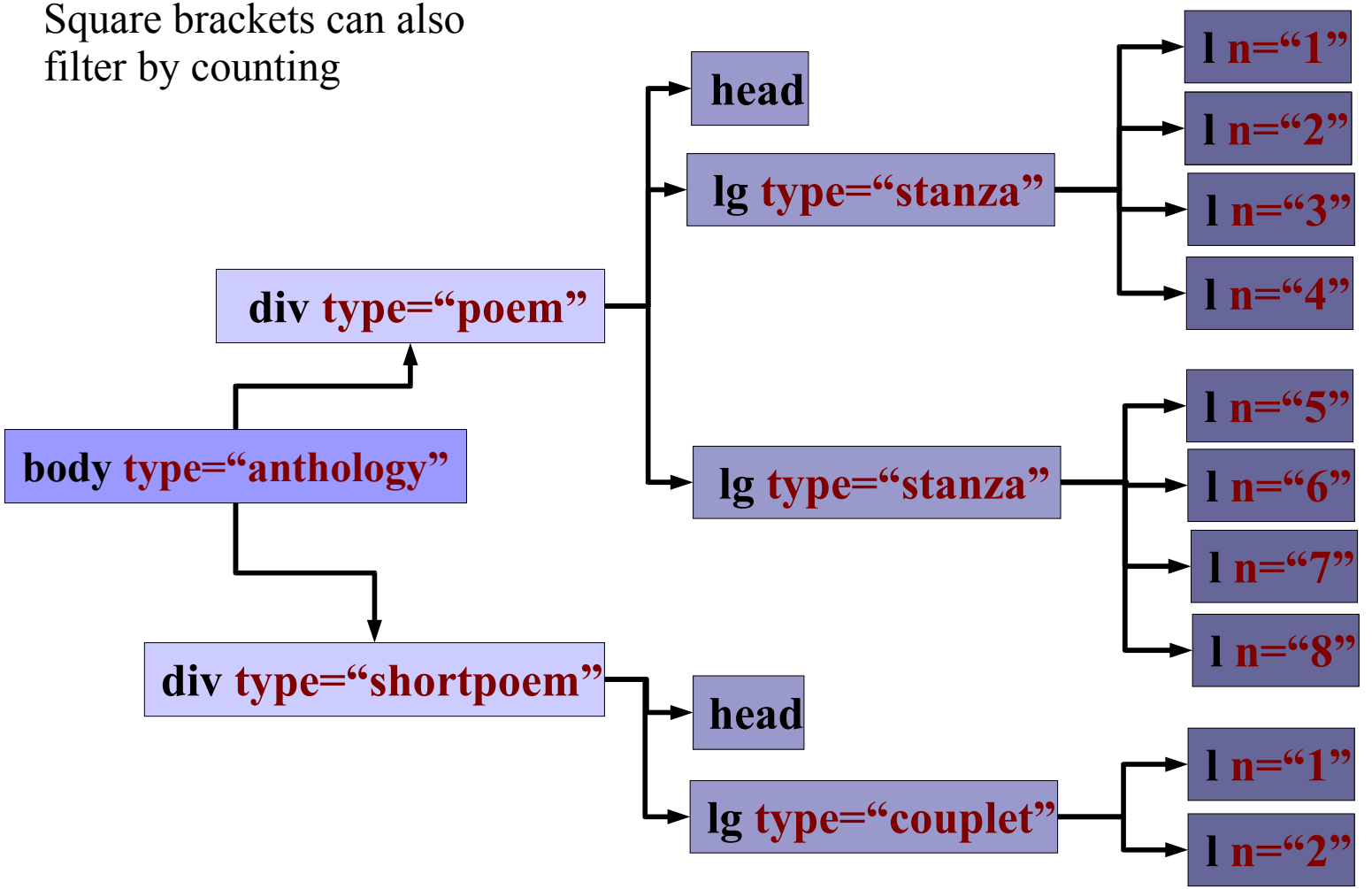


//div[@type="poem"]//l

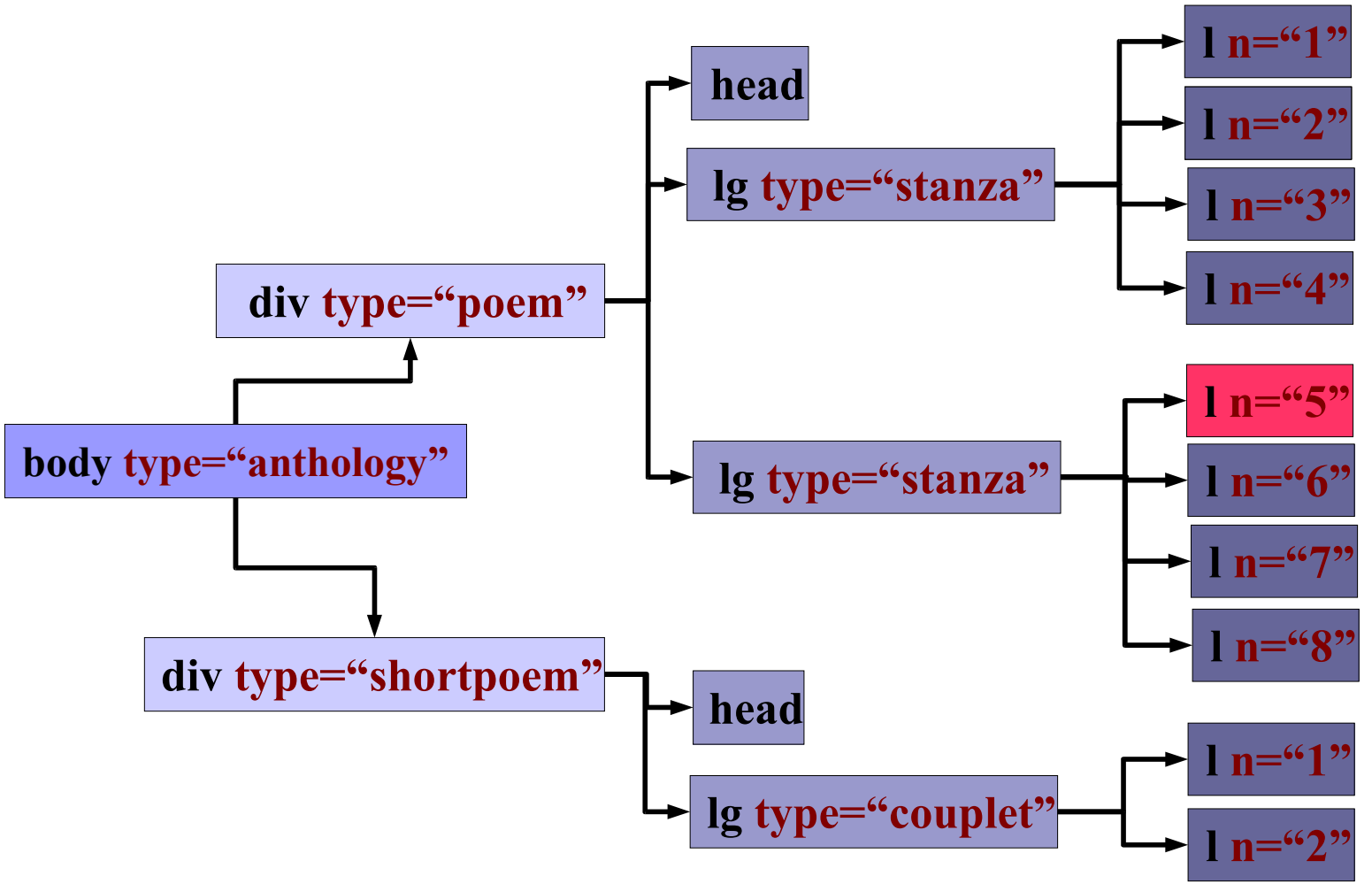


//l[5] ?

Square brackets can also filter by counting



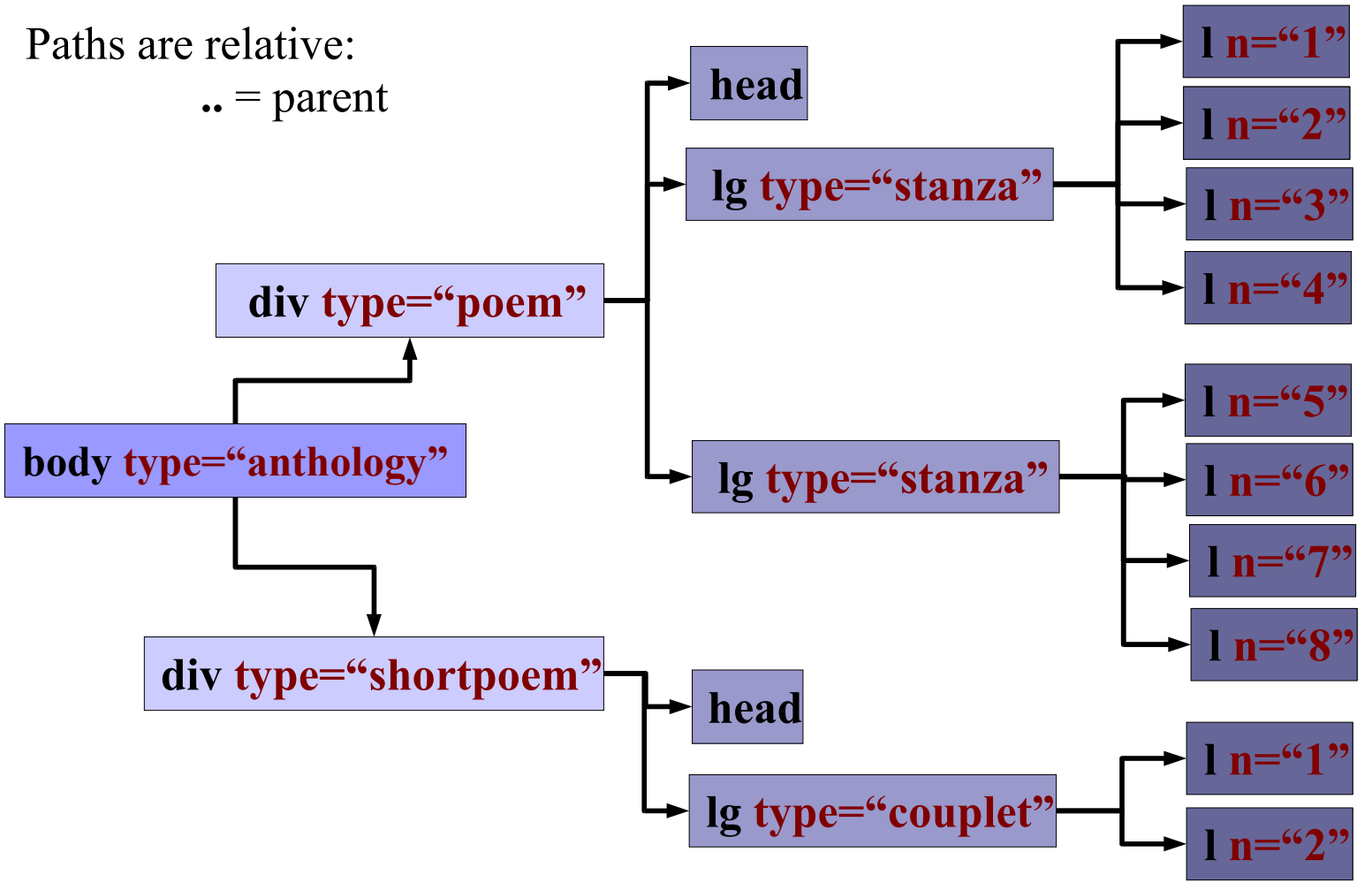
//l[5]


body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

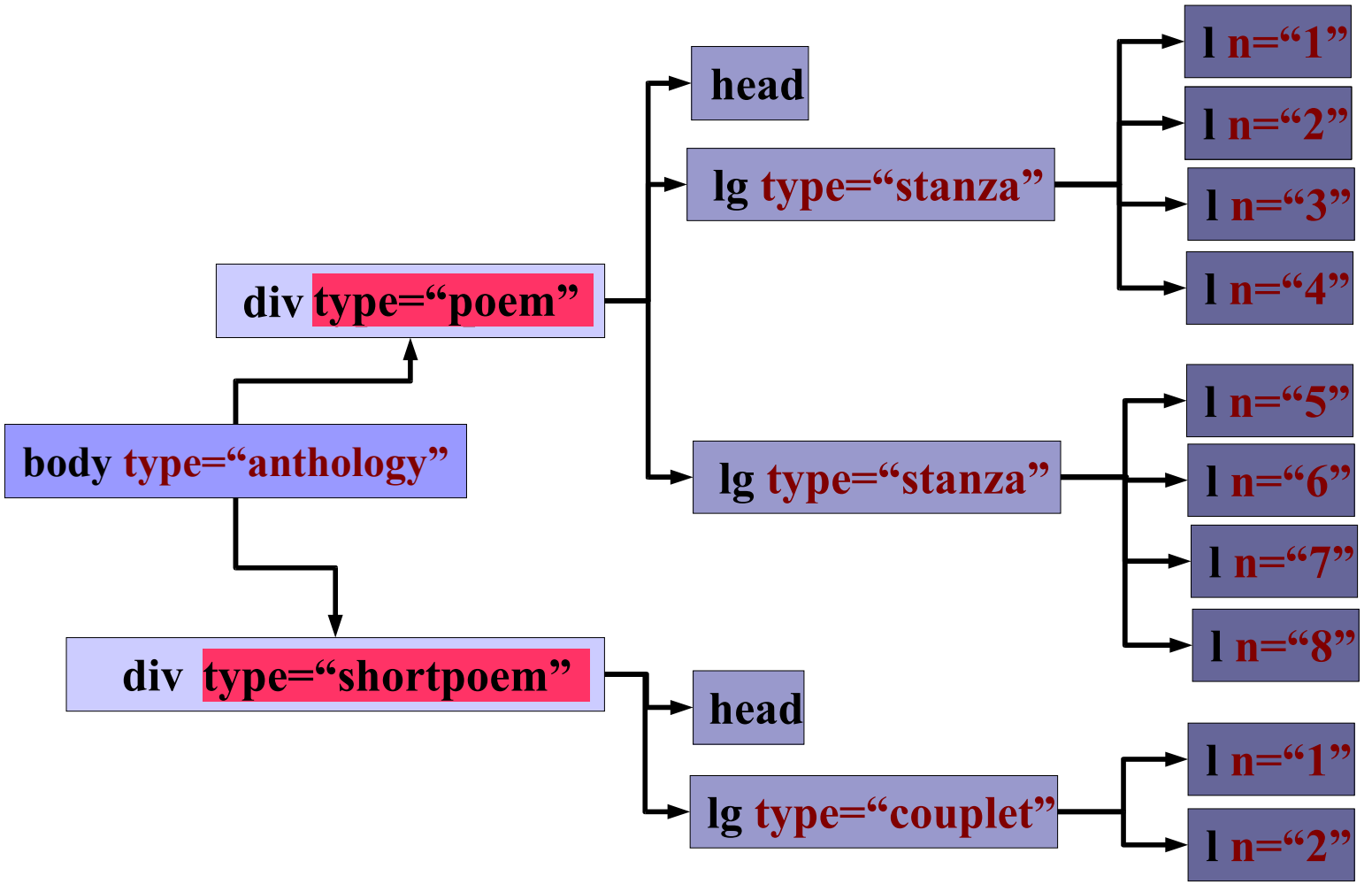
//lg/../@type ?

Paths are relative:

.. = parent

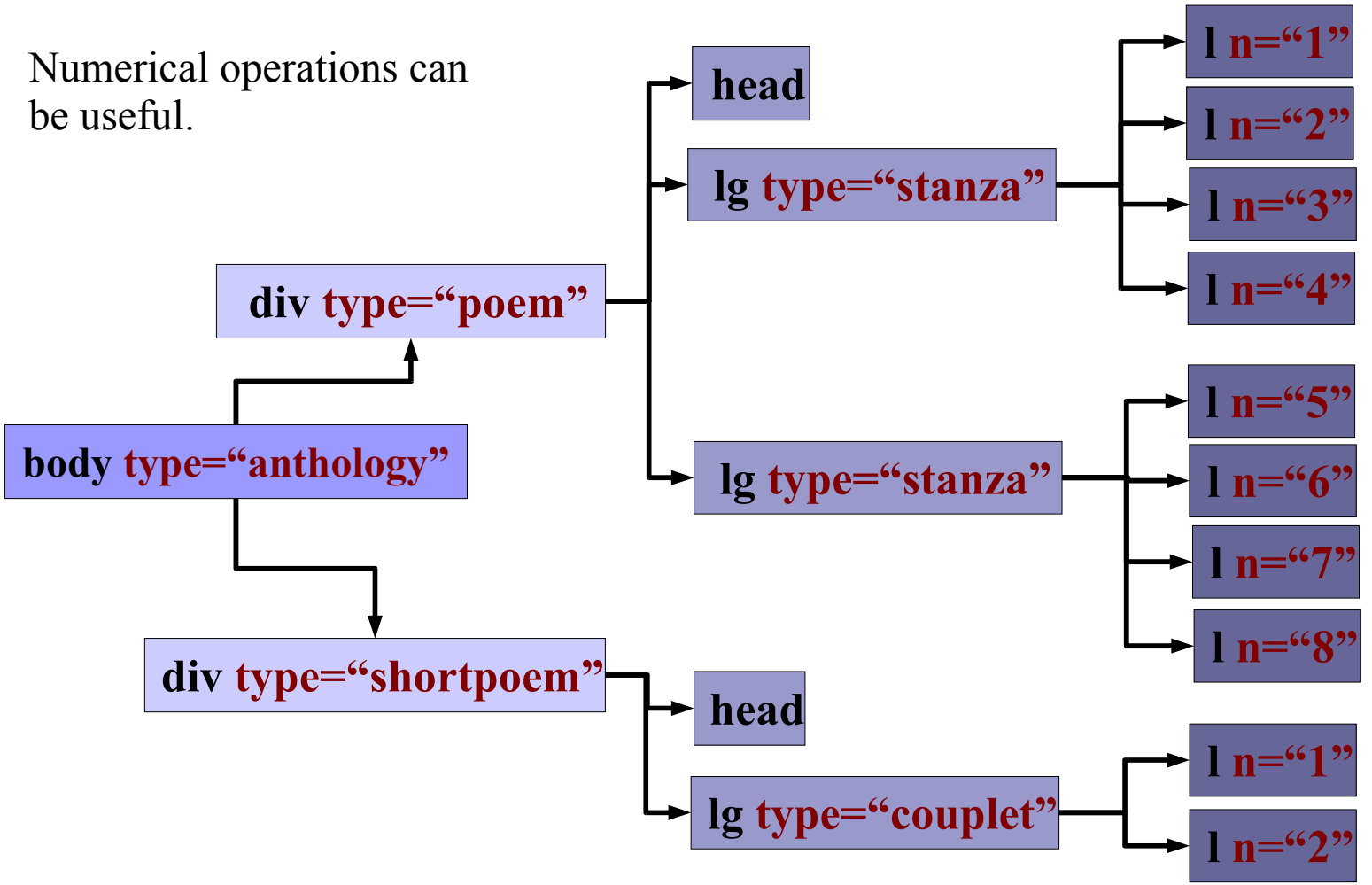


//lg/../@type

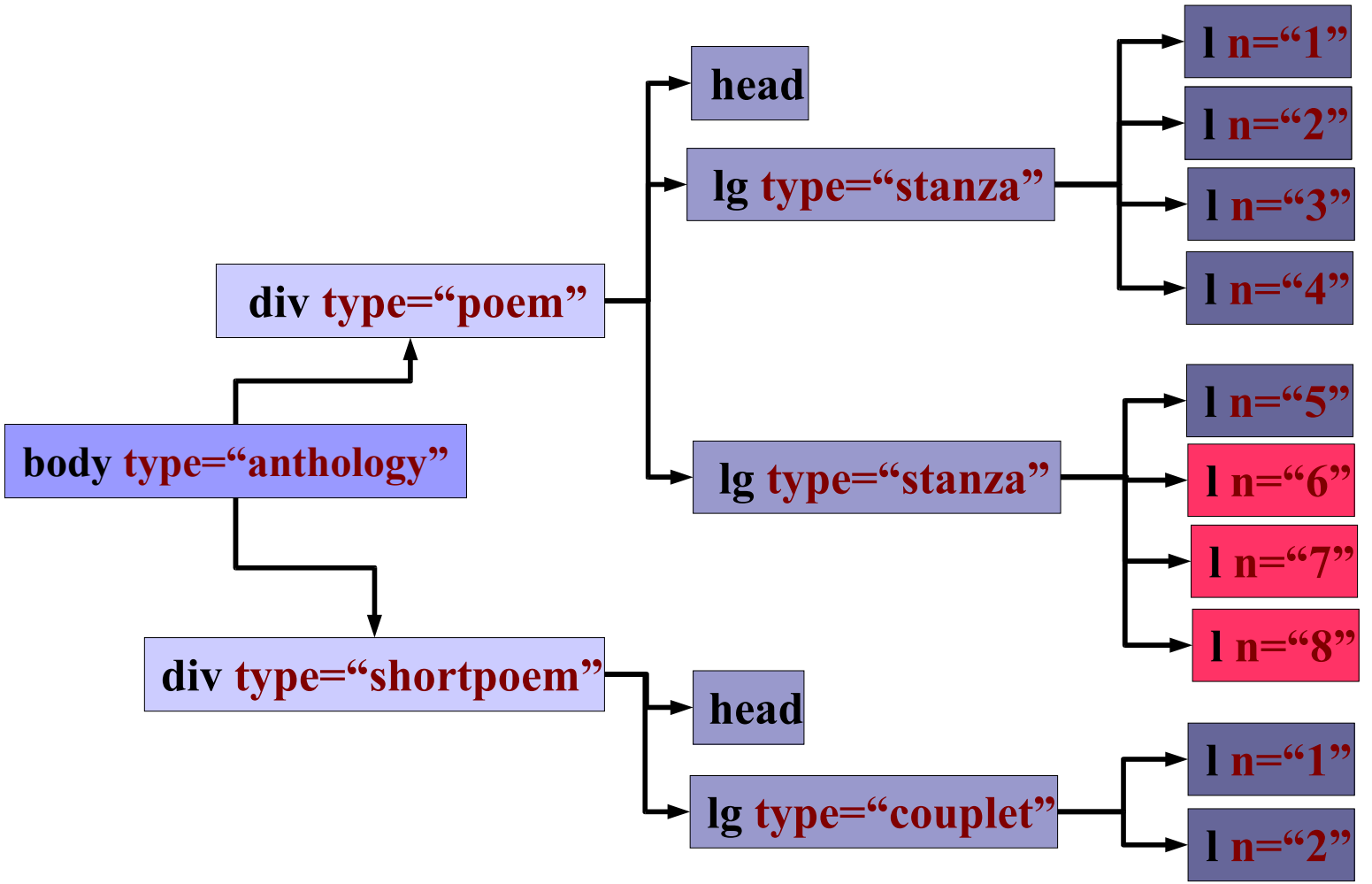

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

//l[@n > 5] ?

Numerical operations can be useful.

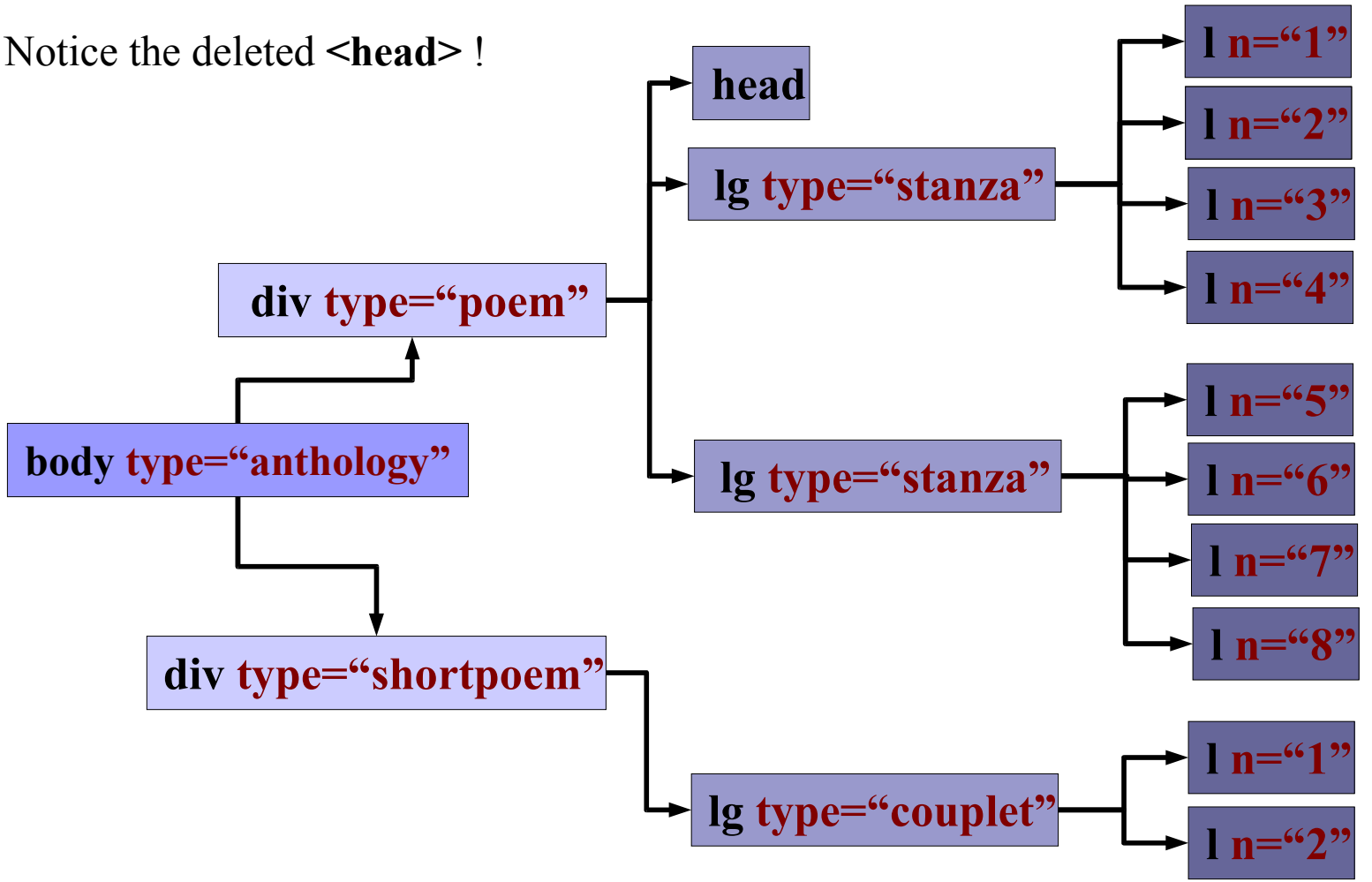


//l[@n > 5]

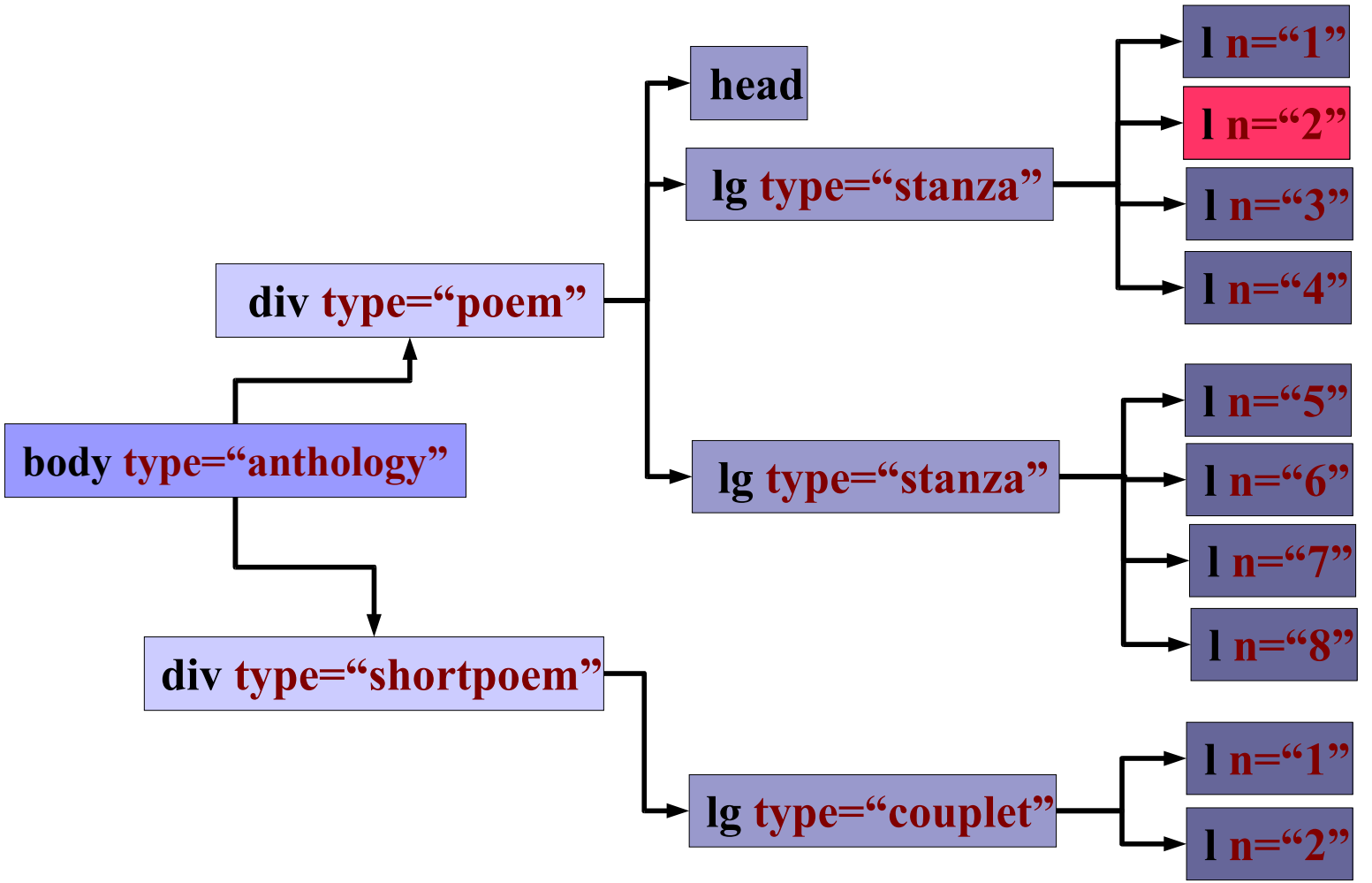

body type="anthology"
div type="poem"
head
lg type="stanza"
l n="1"
l n="2"
l n="3"
l n="4"
lg type="stanza"
l n="5"
l n="6"
l n="7"
l n="8"
div type="shortpoem"
head
lg type="couplet"
l n="1"
l n="2"

//div[head]/lg/l[@n="2"] ?

Notice the deleted **<head>** !

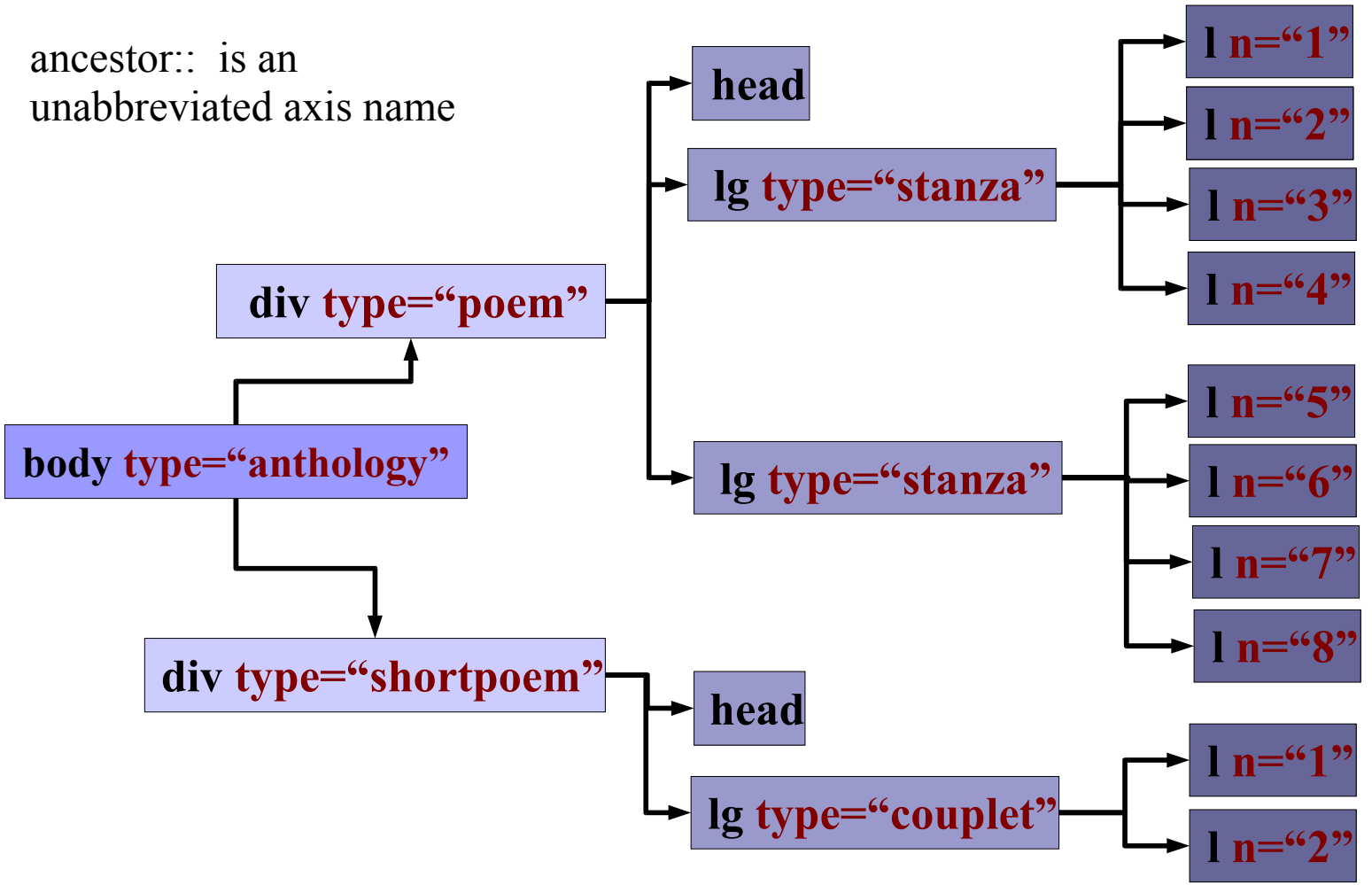


//div[head]/lg/l[@n="2"]

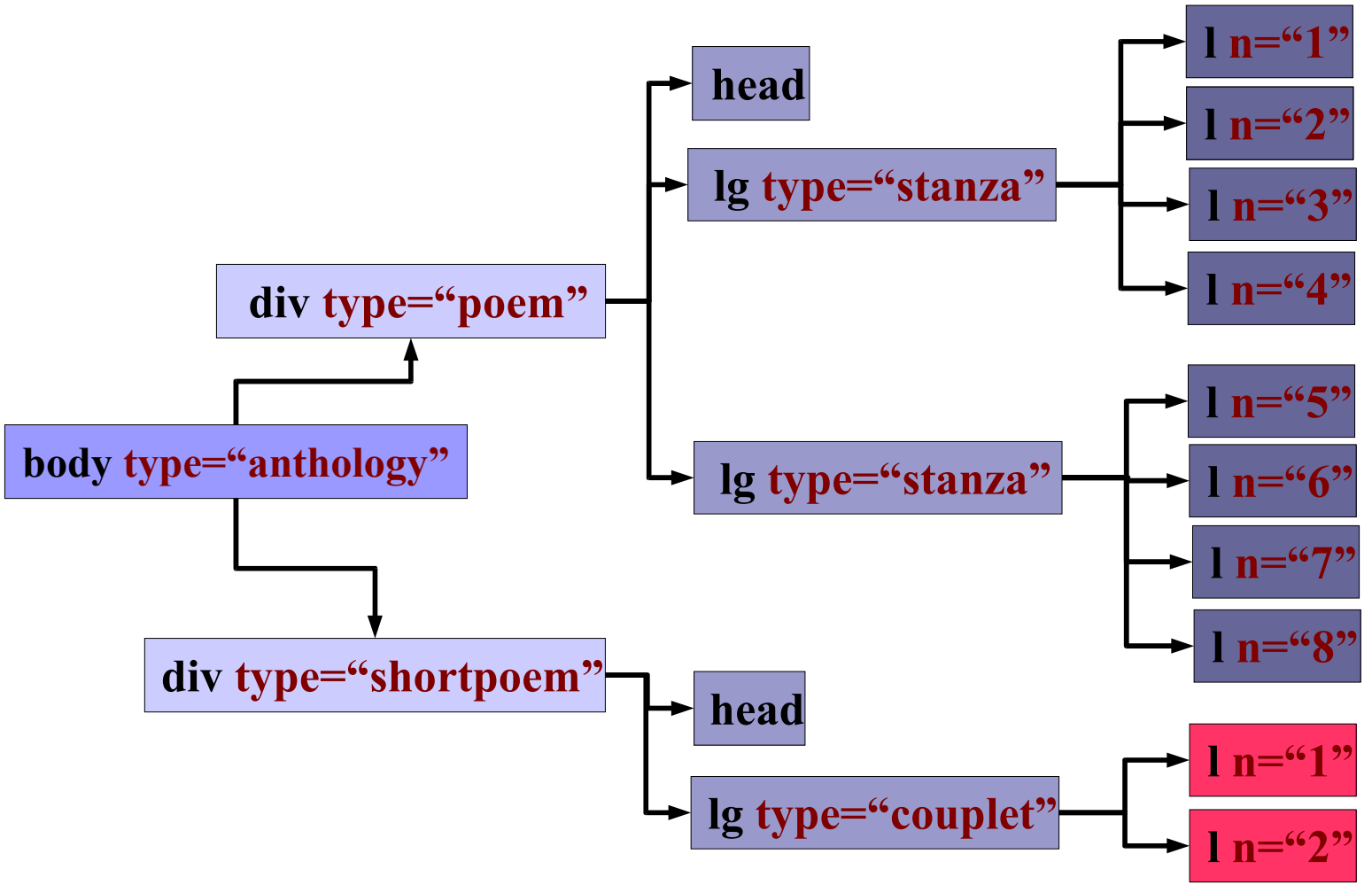


//l[ancestor::div/@type="shortpoem"] ?

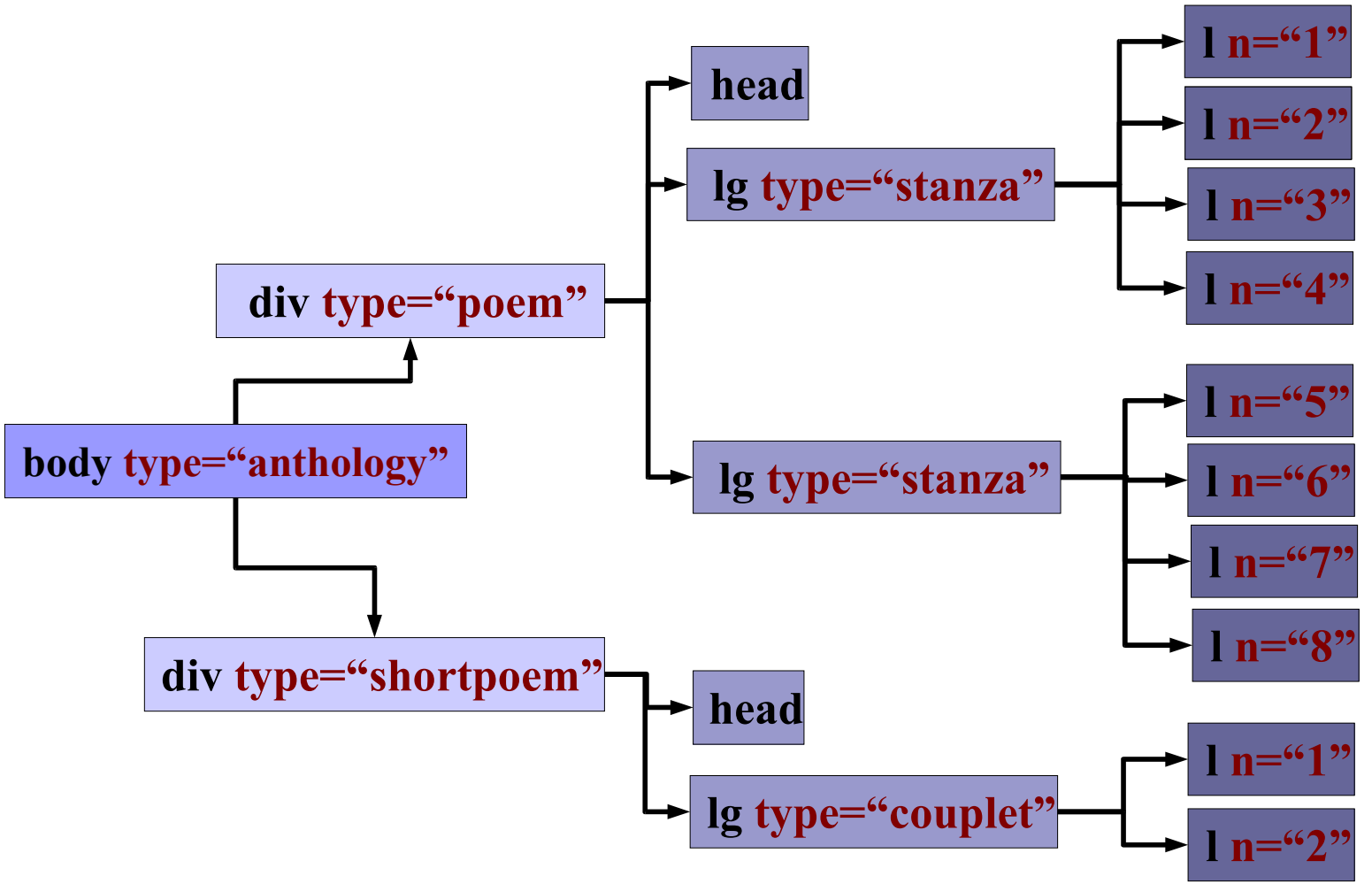
ancestor:: is an unabbreviated axis name



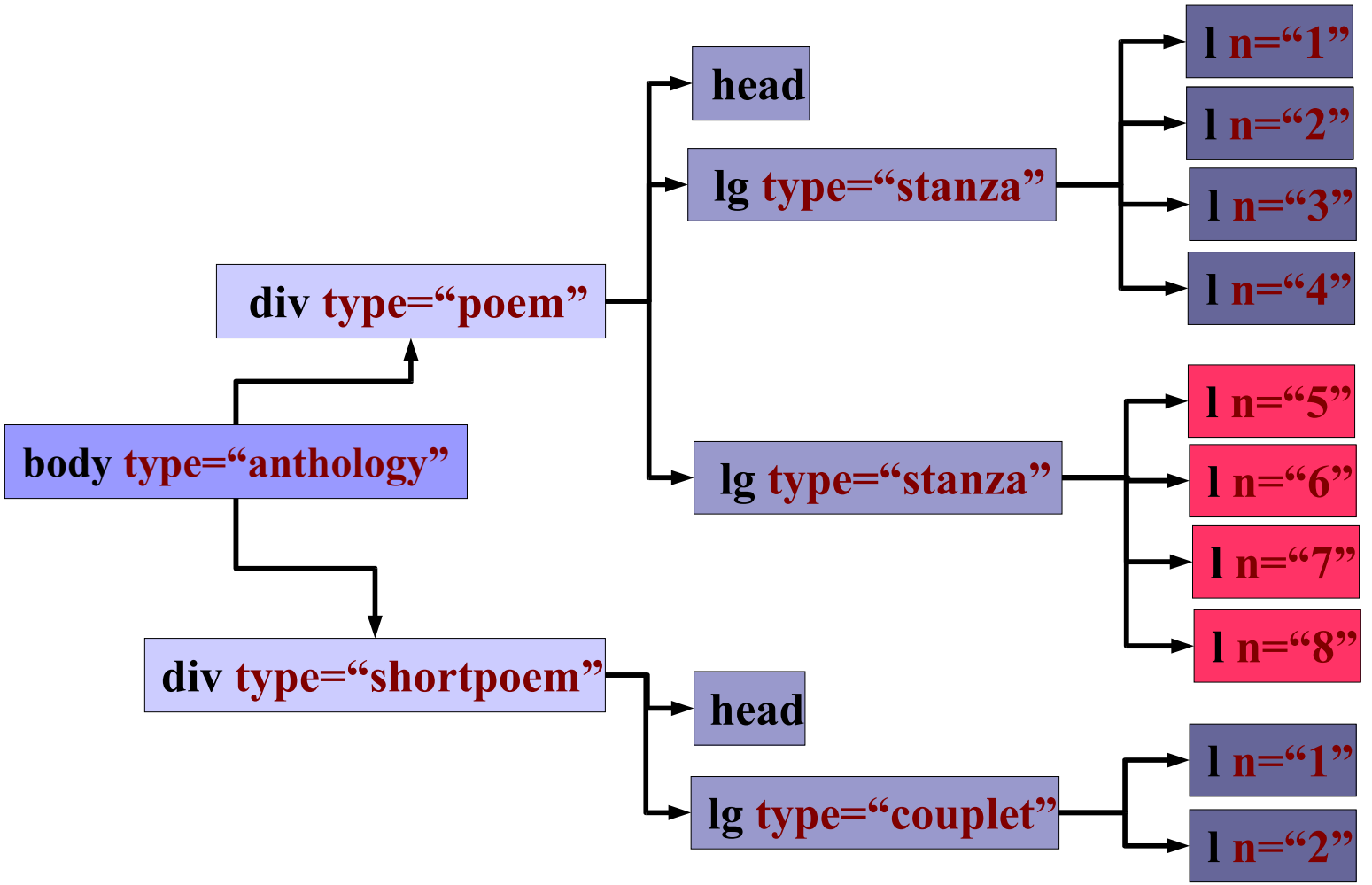
//l[ancestor::div/@type="shortpoem"]



lg[1]/following-sibling::l ?



lg[1]/following-sibling::l



XPath: More About Paths

- A location path results in a node-set
- Paths can be absolute (`/div/lg1/l`)
- Paths can be relative (`l/../../head`)
- Formal Syntax: (`axisname::nodetest[predicate]`)
- For example: `child::div[contains(head, 'ROSE')]`

XPath: Axes

ancestor:: Contains all ancestors (parent, grandparent, etc.) of the current node

ancestor-or-self:: Contains the current node plus all its ancestors (parent, grandparent, etc.)

attribute:: Contains all attributes of the current node

child:: Contains all children of the current node

descendant:: Contains all descendants (children, grandchildren, etc.) of the current node

descendant-or-self:: Contains the current node plus all its descendants (children, grandchildren, etc.)

XPath: Axes (2)

following:: Contains everything in the document after the closing tag of the current node

following-sibling:: Contains all siblings after the current node

parent:: Contains the parent of the current node

preceding:: Contains everything in the document that is before the starting tag of the current node

preceding-sibling:: Contains all siblings before the current node

self:: Contains the current node

Axis examples

- `ancestor::lg` = all `<lg>` ancestors
- `ancestor-or-self::div` = all `<div>` ancestors or current
- `attribute::n` = n attribute of current node
- `child::l` = `<l>` elements directly under current node
- `descendant::l` = `<l>` elements anywhere under current node
- `descendant-or-self::div` = all `<div>` children or current
- `following::lg` = all following `<lg>` elements
- `following-sibling::l` = next `<l>` element at this level
- `parent::lg` = immediate parent `<lg>` element
- `preceding::lg` = all preceding `<lg>` elements
- `preceding-sibling::l` = previous `<l>` element at this level

XPath: Predicates

- `child::lg[attribute::type='stanza']`
- `child::l[@n=4]`
- `child::div[position()=3]`
- `child::div[4]`
- `child::l[last()]`
- `child::lg[last()-1]`

XPath: Abbreviated Syntax

- nothing is the same as `child::`, so `lg` is short for `child::lg`
- @ is the same as `attribute::`, so `@type` is short for `attribute::type`
- . is the same as `self::`, so `./head` is short for `self::node()/head`
- .. is the same as `parent::`, so `../lg` is short for `parent::node()/child::lg`
- // is the same as `descendant-or-self::`, so `div//l` is short for `child::div/descendant-or-self::node()/child`

XPath: Operators

XPath has support for numerical, equality, relational, and boolean expressions

- + (Addition): 3 + 2 = 5
- - (Subtraction): 10 - 2 = 8
- * (Multiplication): 6 * 4 = 24
- div (Division): 8 div 4 = 2
- mod (Modulus): 5 mod 2 = 1
- = (Equal): @age = "74" True (if @age does equal "74")
- or (Boolean OR): @age = '74' or @age = '64' True

XPath Functions: Node-Set Functions

- `count()` Returns the number of nodes in a node-set: `count(person)`
- `id()` Selects elements by their unique ID : `id('S3')`
- `last()` Returns the position number of the last node : `person[last()]`
- `name()` Returns the name of a node: `//*[name('person')]`
- `namespace-uri()` Returns the namesparce URI of a specified node: `namespace-uri(persName)`
- `position()` Returns the position in the node list of the node that is currently being processed : `//person[position()='6']`

XPath Functions: String Functions

- `concat()` Concatenates its arguments: `concat('http://', $domain, '/', $file, '.html')`
- `contains()` Returns true if the second string is contained withing the first string: `//persName[contains(surname, 'van')]`
- `normalize-space()` Removes leading and trailing whitespace and replaces all internal whitespace with one space: `normalise-space(surname)`
- `starts-with()` Returns true if the first string starts with the second: `starts-with(surname, 'van')`
- `string()` Converts the argument to a string: `string(@age)`

XPath Functions: String Functions (2)

- `substring` Returns part of a string of specified start character and length: `substring(surname, 5,4)`
- `substring-after()` Returns the part of the string that is after the string given:
  `substring-after(surname, 'De')`
- `substring-before` Returns the part of the string that is before the string given:
  `substring-before(@date, '-')`
- `translate()` Performs a character by character replacement. It looks at the characters in the first string and replaces each character in the first argument by the correspondin one in the second argument:
  `translate('1234', '24', '68')`

XPath Functions: Numeric Functions

- `ceiling()` Returns the smallest integer that is not less that the number given: `ceiling(3.1415)`
- `floor()` Returns the largest integer that is not greater than the number given: `floor(3.1415)`
- `number()` Converts the input to a number: `number('100')`
- `round()` Rounds the number to the nearest integer: `round(3.1415)`
- `sum()` Returns the total value of a set of numeric arguments: `sum(//person/@age)`
- `not()` Returns true if the condition is false: `not(position() >5)`

XPath: Where can I use XPath?

Learning all these functions, though a bit tiring to begin with, can be very useful as they are used throughout XML technologies, but especially in XSLT and XQuery.

What is the XSL family?

- XPath: a language for expressing paths through XML trees
- XSLT: a programming language for transforming XML
- XSL FO: an XML vocabulary for describing formatted pages

XSLT

The XSLT language is

- Expressed in XML; uses namespaces to distinguish output from instructions
- Purely functional
- Reads and writes XML trees
- Designed to generate XSL FO, but now widely used to generate HTML

How is XSLT used? (1)

- With a command-line program to transform XML (eg to HTML)
  - Downside: no dynamic content, user sees HTML
  - Upside: no server overhead, understood by all clients
- In a web server *servlet*, eg serving up HTML from XML (eg Cocoon, Axkit)
  - Downside: user sees HTML, server overhead
  - Upside: understood by all clients, allows for dynamic changes

How is XSLT used? (2)

- In a web browser, displaying XML on the fly
  - Downside: many clients do not understand it
  - Upside: user sees XML
- Embedded in specialized program
- As part of a chain of production processes, performing arbitrary transformations

XSLT implementations

MSXML	Built into Microsoft Internet Explorer
Saxon	Java-based, standards leader (basic version free)
Xerces	Java-based, widely used in servlets (open source)
libxslt	C-based, fast and efficient (open source)
transformiix	C-based, used in Mozilla (open source)

What do you mean, ‘transformation’?

Take this

```
<recipe>
 <title>Pasta for beginners</title>
 <ingredients>
  <item>Pasta</item>
  <item>Grated cheese</item>
 </ingredients>
 <cook>Cook the pasta and mix with the cheese</cook>
</recipe>
```

and make this

```
<html>
 <h1>Pasta for beginners</h1>
 <p>Ingredients:  Pasta Grated cheese</p>
 <p>Cook the pasta and mix with the cheese</p>
</html>
```

How do you express that in XSL?


```
<xsl:stylesheet version="1.0">
 <xsl:template match="recipe">
  <html>
   <h1>
    <xsl:value-of select="title"/>
   </h1>
   <p>Ingredients:
     <xsl:apply-templates select="ingredients/item"/>
   </p>
   <p>
    <xsl:value-of select="cook"/>
   </p>
  </html>
 </xsl:template>

</xsl:stylesheet>
```

Structure of an XSL file


```
<xsl:stylesheet version="1.0">
 <xsl:template match="tei:div">
 </xsl:template>
 <xsl:template match="tei:p">
 </xsl:template>

</xsl:stylesheet>
```

The `tei:div` and `tei:p` are *XPath expressions*, which specify which bit of the document is matched by the template.

Any element not starting with xsl: in a template body is put into the output.

The Golden Rules of XSLT

1. If there is no template matching an element, we process the elements inside it
2. If there are no elements to process by Rule 1, any text inside the element is output
3. Children elements are not processed by a template unless you explicitly say so
4. ```
   xsl:apply-templates select="XX"
   ```

   looks for templates which match element "XX";

   ```
   xsl:value-of select="XX"
   ```

   simply gets any text from that element
5. The order of templates in your program file is immaterial
6. You can process any part of the document from any template
7. Everything is well-formed XML. Everything!
   ```

Building a TEI stylesheet (1)

Process everything in the document and make an HTML document:

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

</xsl:template>
```

but ignore the `<teiHeader>`

```
<xsl:template match="tei:TEI">
 <xsl:apply-templates select="tei:text"/>

</xsl:template>
```

and do the `<front>` and `<body>` separately

```
<xsl:template match="tei:text">
 <h1>FRONT MATTER</h1>
 <xsl:apply-templates select="tei:front"/>
 <h1>BODY MATTER</h1>
 <xsl:apply-templates select="tei:body"/>

</xsl:template>
```

Building a TEI stylesheet (2)

Templates for paragraphs and headings:

```
<xsl:template match="tei:p">
 <p>
  <xsl:apply-templates/>
 </p>
</xsl:template>
<xsl:template match="tei:div">
 <h2>
  <xsl:value-of select="tei:head"/>
 </h2>
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="tei:div/tei:head"/>
```

Notice how we avoid getting the heading text twice.

Why did we need to qualify it to deal with just `<head>` inside `<div>`?

Building a TEI stylesheet (3)

Now for the lists. We'll need to look at the 'type' attribute to decide what sort of HTML list to produce:

```
<xsl:template match="tei:list">
 <xsl:choose>
  <xsl:when test="@type='ordered'">
   <ol>
    <xsl:apply-templates/>
   </ol>
  </xsl:when>
  <xsl:when test="@type='unordered'">
   <ul>
    <xsl:apply-templates/>
   </ul>
  </xsl:when>
  <xsl:when test="@type='gloss'">
   <dl>
    <xsl:apply-templates/>
   </dl>
  </xsl:when>
 </xsl:choose>
</xsl:template>
```

Building a TEI stylesheet (5)

It would be nice to get those sections numbered, so let's change the template and let XSLT do it for us:

```
<xsl:template match="tei:div">
 <h2>
  <xsl:number level="multiple" count="tei:div"/>
  <xsl:text>.  </xsl:text>
  <xsl:value-of select="tei:head"/>
 </h2>
 <xsl:apply-templates/>

</xsl:template>
```

Building a TEI stylesheet (6)

and number the paragraphs as well

```
<xsl:template match="tei:p">
 <p>
  <xsl:number/>
  <xsl:text>:  </xsl:text>
  <xsl:apply-templates/>
 </p>
</xsl:template>
```

Building a TEI stylesheet (7)

Let's summarize all the manuscripts, *sorting* them by repository and ID number

```
<xsl:template match="tei:TEI">
 <ul>
  <xsl:for-each select=".//tei:msDescription">
   <xsl:sort select="tei:msIdentifier/tei:repository"/>
   <xsl:sort select="tei:msIdentifier/tei:idno"/>
   <li>
    <xsl:value-of select="tei:msIdentifier/tei:repository"/>:
     <xsl:value-of select="tei:msIdentifier/tei:settlement"/>:
     <xsl:value-of select="tei:msIdentifier/tei:idno"/>
   </li>
  </xsl:for-each>
 </ul>
</xsl:template>
```

Modes

You can process the same elements in different ways using modes:

```
<xsl:template match="/">
 <xsl:apply-templates select=".//tei:div" mode="toc"/>
 <xsl:apply-templates/>
</xsl:template>
<xsl:template match="tei:div" mode="toc">
  Heading <xsl:value-of select="tei:head"/>
</xsl:template>
```

This is a very useful technique when the same information is processed in different ways in different places.

Named templates, parameters and variables

`<xsl:template name="...">`: define a named template
`<xsl:call-template>`: call a named template
`<xsl:param>`: specify a parameter in a template definition
`<xsl:with-param>`: specify a parameter when calling a template
`<xsl:variable name="...">`: define a variable

Variables


```
<xsl:template match="tei:p">
 <xsl:variable name="n">
  <xsl:number/>
 </xsl:variable>
  Paragraph <xsl:value-of select="$n"/>
 <a name="P$n"/>
 <xsl:apply-templates/>

</xsl:template>
```

Named templates


```
<xsl:template match="tei:div">
 <html>
  <xsl:call-template name="header">
   <xsl:with-param name="title" select="tei:head"/>
  </xsl:call-template>
  <xsl:apply-templates/>
 </html>
</xsl:template>
<xsl:template name="header">
 <xsl:param name="title"/>
 <head>
  <title>
   <xsl:value-of select="$title"/>
  </title>
 </head>
</xsl:template>
```

Top-level commands

`<xsl:import href="...">`: include a file of XSLT templates, overriding them as needed

`<xsl:include href="...">`: include a file of XSLT templates, but do not override them

`<xsl:output>`: specify output characteristics of this job

Some useful `xsl:output` attributes

`method="`xml | html | text"
`encoding="`*string"*
`omit-xml-declation="`yes | no"
`doctype-public="`*string"*
`doctype-system="`*string"*
`indent="`yes | no"

An identity transform


```
<xsl:output method="xml" indent="yes" encoding="iso-8859-1" do
<xsl:template match="/">
 <xsl:copy-of select="."/>

</xsl:template>
```

A near-identity transform


```
<xsl:template match="*|@*|processing-instruction()">
 <xsl:copy>
  <xsl:apply-templates select="*|@*|processing-instruction()|co
 </xsl:copy>
</xsl:template>
<xsl:template match="text()">
 <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="tei:p">
 <para>
 <xsl:apply-templates/>
 </para>
</xsl:template>
```

TEI Stylesheets

A library of stylesheets for transforming TEI documents to

- HTML
- XSL Formatting Objects
- LaTeX

at http://www.tei-c.org/Stylesheets, with a form-filling interface for the HTML at http://www.tei-c.org/tei-bin/stylebear (also on your CD)

TEI Stylesheets (example)

An example of an importing stylesheet:

```
<xsl:stylesheet version="1.0">
 <xsl:import href="/usr/share/xml/tei/stylesheet/html/tei.xsl"/
 <xsl:param name="splitLevel">1</xsl:param>
 <xsl:param name="numberHeadings"/>
 <xsl:param name="topNavigationPanel">true</xsl:param>
 <xsl:param name="bottomNavigationPanel">true</xsl:param>

</xsl:stylesheet>
```

A more complex example (the TEI site)


```
 <xsl:param name="oddmode">html</xsl:param>
 <xsl:variable name="top" select="/"/>
 <xsl:param name="STDOUT">true</xsl:param>
 <xsl:param name="alignNavigationPanel">left</xsl:param>
 <xsl:param name="authorWord"/>
 <xsl:param name="autoToc"/>
 <xsl:param name="bottomNavigationPanel">true</xsl:param>
 <xsl:param name="cssFile">Stylesheets/tei.css</xsl:param>
 <xsl:param name="feedbackURL">http://www.tei-c.org/Consortium/T
 <xsl:param name="feedbackWords">Contact</xsl:param>
 <xsl:param name="homeURL">http://www.tei-c.org/</xsl:param>
 <xsl:param name="homeWords">TEI Home</xsl:param>
 <xsl:param name="institution">Text Encoding
Initiative</xsl:param>
 <xsl:param name="leftLinks">true</xsl:param>
 <xsl:param name="searchURL">http://search.ox.ac.uk/web/related/
 <xsl:param name="searchWords">Search this site</xsl:param>
 <xsl:param name="showTitleAuthor">1</xsl:param>
 <xsl:param name="subTocDepth">-1</xsl:param>
 <xsl:param name="topNavigationPanel"/>
 <xsl:param name="numberHeadings">true</xsl:param>
 <xsl:template name="copyrightStatement">Copyright TEI
```

Result on TEI web site

The Text Encoding Initiative

TEI: Yesterday's information tomorrow

Home | Guidelines | Projects | Tutorials | Software | History | FAQs | P5 | Consortium | Activities | SIGs | Wiki | Join in/Contact | Members area

Search

The Text Encoding Initiative (TEI) Guidelines are an international and interdisciplinary standard that facilitates libraries, museums, publishers, and individual scholars represent a variety of literary and linguistic texts for online research, teaching, and preservation.

The TEI standard is maintained by a Consortium of leading Institutions and Projects worldwide. Information on projects which use the TEI, who is a member, and how to join, can all be found via the links above. Consortium members contribute to its financial stability and elect members to its Council and Board.

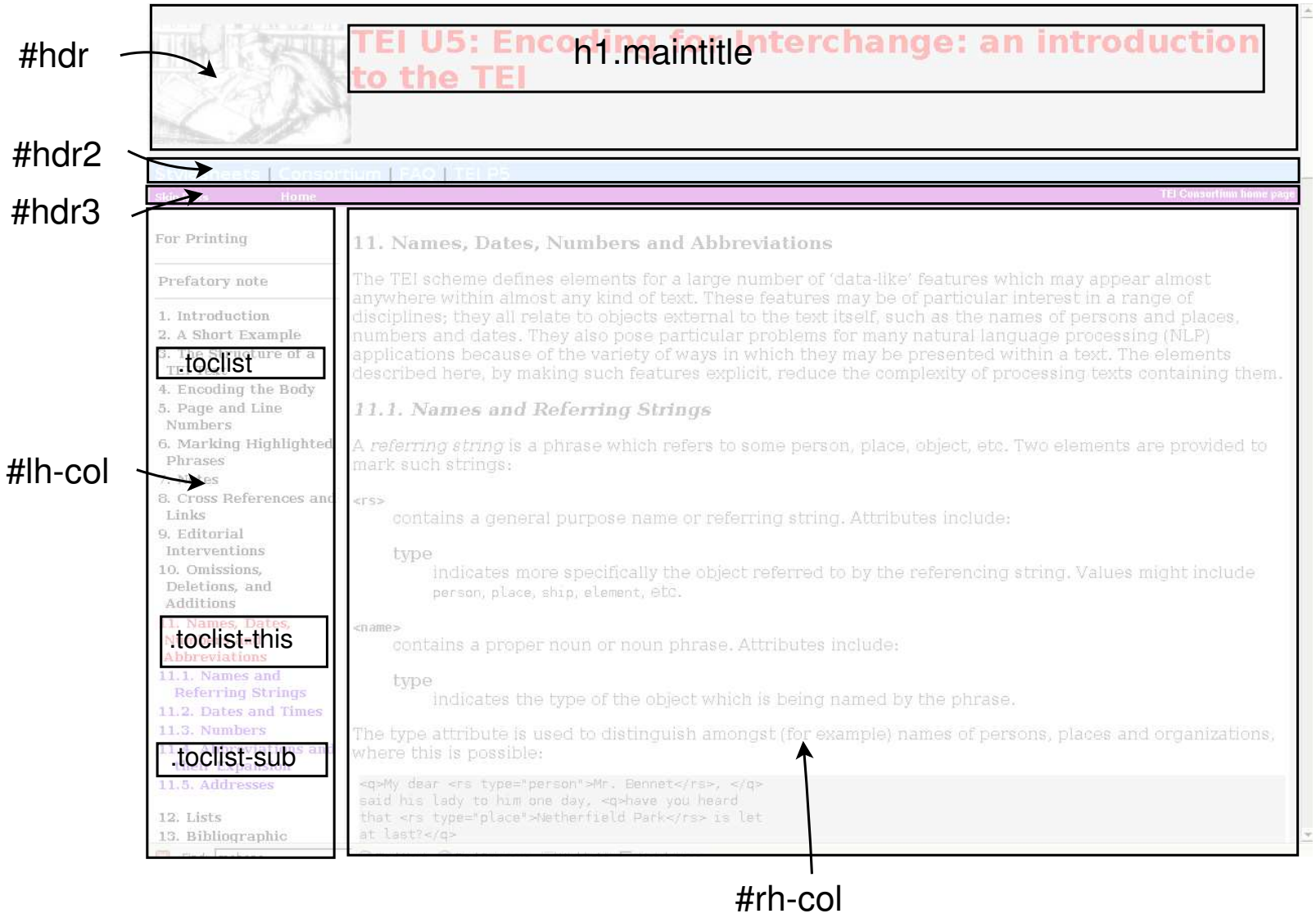
The Guidelines are the chief deliverable of the TEI Consortium, along with a range of tutorials, case studies, presentations, and software developed for or adapted to the TEI. The latest release of the Guidelines under development is P5.

The TEI was originally sponsored by the Association of Computers in the Humanities (ACH), the Association for Computational Linguistics (ACL), and the Association of Literary and Linguistic Computing (ALLC). Major support has been received from the U.S. National Endowment for the Humanities (NEH), the European Community, the Mellon Foundation, and the Social Science and Humanities Research Council of Canada.

Want to become active in the TEI Community? Join a Special Interest Group, sign up for the mailing list, and come to our annual meetings.

The fifth Annual Members Meeting will be held 28-29 October 2005, at the Bulgarian Academy of Sciences, Sofia, Bulgaria and is open to members and non-members alike. (see announcement for program and registration details.)

Skeleton of result



Skeleton of result (2)


```
<div id="hdr">
  <xsl:call-template name="hdr"/>
</div>
```

```
<div id="hdr2"> <xsl:call-template name="hdr2"/> </div>
```

```
<div id="hdr3"> <xsl:call-template name="hdr3"/> </div>
```

```
<div id="lh-col">
<div id="lh-col-top">
<xsl:call-template
name="lh-col-top"/>
</div>
<div id="lh-col-bottom">
<xsl:call-template
name="lh-col-bottom">
<xsl:with-param
name="currentID"
select="$currentID"/>
</xsl:call-template>
</div>
</div>
```

```
<div id="rh-col">
  <div id="rh-col-top">
    <xsl:call-template name="rh-col-top"/>
  </div>
  <div id="rh-col-bottom">
    <xsl:call-template name="rh-col-bottom">
      <xsl:with-param name="currentID" select="$currentID"/>
    </xsl:call-template>
  </div>
</div>
<div id="lh-col">
  <div id="lh-col-top">
    <xsl:call-template name="lh-col-top"/>
  </div>
  <div id="lh-col-bottom">
    <xsl:call-template name="lh-col-bottom">
      <xsl:with-param name="currentID" select="$currentID"/>
    </xsl:call-template>
  </div>
</div>
```

What is XSLT good for?

Yes, it is useful for making web pages from TEI texts, but it is also a good tool for

- Selecting subsets of our texts for further processing
- Summarizing aspects of our texts
- Checking our text in ways that DTDs and schemas cannot
- Converting text into formats other than HTML or XSL FO

we can solve many of the problems with a small range of techniques.

Finding any occurrence of an element

Very often, we will sit on the root element and process all the occurrences of a specific element by using the **descendant** axis:

```
<xsl:template match="/">
 <html>
  <body>
    Pages:
    <xsl:value-of select="count(descendant::tei:pb)"/>
  </body>
 </html>
</xsl:template>
```

here we use the **count** function. We continue to generate an HTML document to provide a convenient reporting format.

Converting to other formats

We can write a template to list the size of paragraphs:

```
<xsl:template match="tei:p">
 <xsl:variable name="contents">
  <xsl:apply-templates select=".//text()"/>
 </xsl:variable>
 <xsl:number level="any"/>:
 <xsl:value-of select="string-length($contents)"/>

</xsl:template>
```

Adding `<xsl:output method="text"/>` will produce pure text output which could be loaded into a spreadsheet or database.

XSLT extensions

Although we will not be covering them here, most XSLT processors support various extensions, which will be formalized in version 2.0:

- The ability to create *multiple* output files from one input
- The ability to 'escape' to another language (eg Java) for special purposes
- The ability to turn results into input trees for further processing