XQuery Language

Danilo Montesi

danilo.montesi@unibo.it

http://www.unibo.it/docenti/danilo.montesi

XQuery in brief

- XQuery is a query language for data expressed in XML, and can thus be used to access structured and semi-structured documents.
- It became a W3C Recommendation in January 2007.
- XQuery is a case sensitive language (such as XML), it covers the latest version of XPath and has similar functionality to XSLT (eXtensible Stylesheet Language Transformations).
- We will see the essential aspects (i.e. widely implemented and used) of the current version 3.0. So we do not treat:
  - The definition of functions.
  - Extensions for information retrieval (XQuery-FullText).
  - The documents for updating extensions (XQuery-Update).

Subjects outline

- XQuery data models.
- Navigation expressions (Path Expressions).
- FLWOR expressions.
- Other commonly used expressions: conditional expressions (if-then-else), comparison, arithmetic and logical operators, quantifiers (exists / each).
- Some key standard features.

XQuery Data Model

Sequences 1/2

- Unlike SQL, which operates on relationships, XQuery operates on sequences, which may contain:
  - **Atomic values**, as the string "hello" or the entire 3.
  - **Nodes**.
- An XQuery expression receives zero (in the case of constructors) or more sequences and produces a sequence.
- The main features of the sequences are the following:
  - The sequences are **ordered**, therefore:

    (1, 2) is different from (2, 1).
  - The sequences are **not nested**, therefore:

    ((), 1, (2, 3)) is equal to (1, 2, 3).
  - There is no difference between an *item* and a sequence with the same *item*:

    (1) is equal to 1.

Sequences 2/2

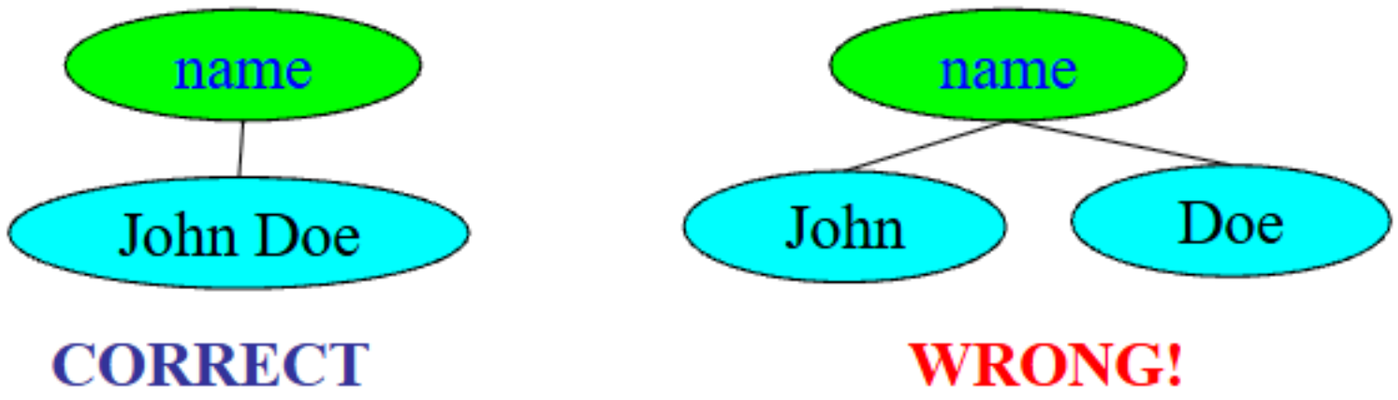
- To manipulate sequences XQuery provides the following operators:
- **,** (comma) and **to** (the following are examples of alternative syntaxes to define the sequence of integers 1, 2 and 3):
  - (1, 2, 3)
  - (1, (), (2, 3))
  - (1 to 3)
  - (1, 2 to 3)
- union (also in its equivalent form |), intersect, except
  - (A) union (A, B) -> (A, B)
  - (A, B) intersect (B, C) -> (B)
  - (A, B) except (B) -> (A)

Nodes 1/3

- In addition to atomic values, a sequence can contain nodes.
- XML documents and fragments are represented as trees, whose nodes can be of type: **document**, **element**, **attribute**, **namespace**, **text**, **comment**, and **processing instruction**.
- The **namespace** nodes such as <xml: lang>, are not represented in the XQuery data model, so we do not consider them.
- In a similar way, we will not treat **comments** and **processing instructions**, which are not (or should not) be used to store data.
- The **text value** (string value) of the nodes of document and element type correspond to the concatenation of the text values of all its descendants textual, in the order in which they are in the document.

Nodes 2/3

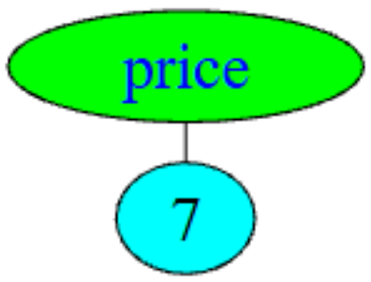
- The nodes of type attribute are not sorted - if extracted using an XQuery query, the order does not necessarily correspond to the one in which they are encountered in the source XML document.
- Once extracted, however, they are inserted in a sequence, of which they keep the order.
- Two **text nodes** can not be adjacent: an element `<name>John Doe</name>` is therefore represented as follows.



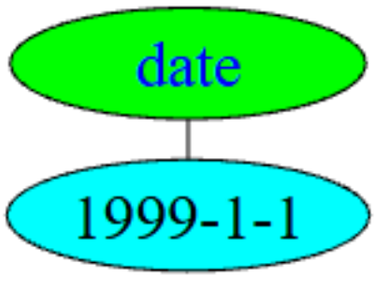
Nodes (3/3)

- If a schema is associated with the document (XML Schema), the nodes can have a type, as in the following examples:

It's of type *string* with the *text value* "T.Harris".



It's of type *integer* with *integer value 7.*

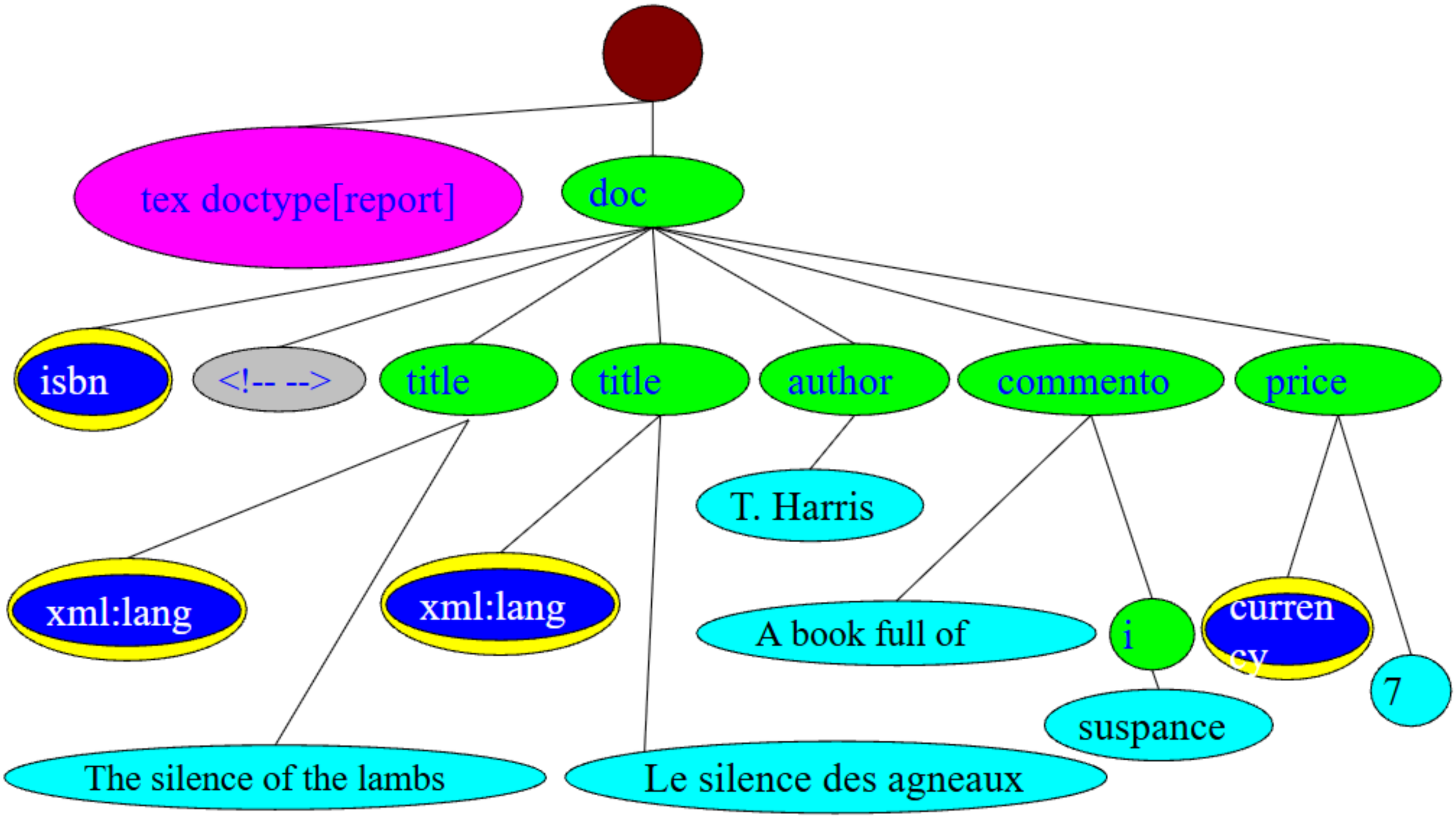


It's of type *date* with value "1st of January 1999".

Examples of nodes 1/2

```
<?xml version="1.0"?>
<?tex doctype[report] ?>
<doc isbn="2-266-04744-2">
    <!-- editor is missing ! -->
    <author>T. Harris</author>
    <title xml:lang="en">The silence of the
lambs</title>
 <title xml:lang="fr">Le silence des agneaux</title>
<commento>
    A book with a lot of <i>suspance</i>
    </commento>
    <price currency="euro">7</price>
</doc>
```

Examples of nodes 2/2



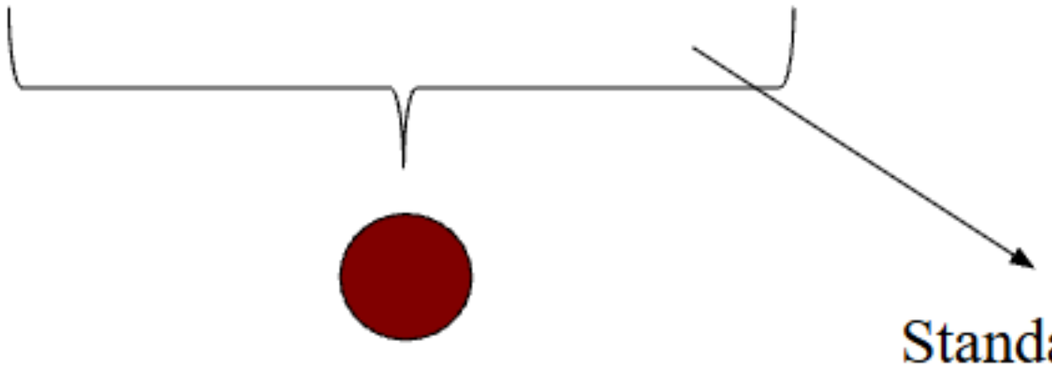
Path Expressions

Observations

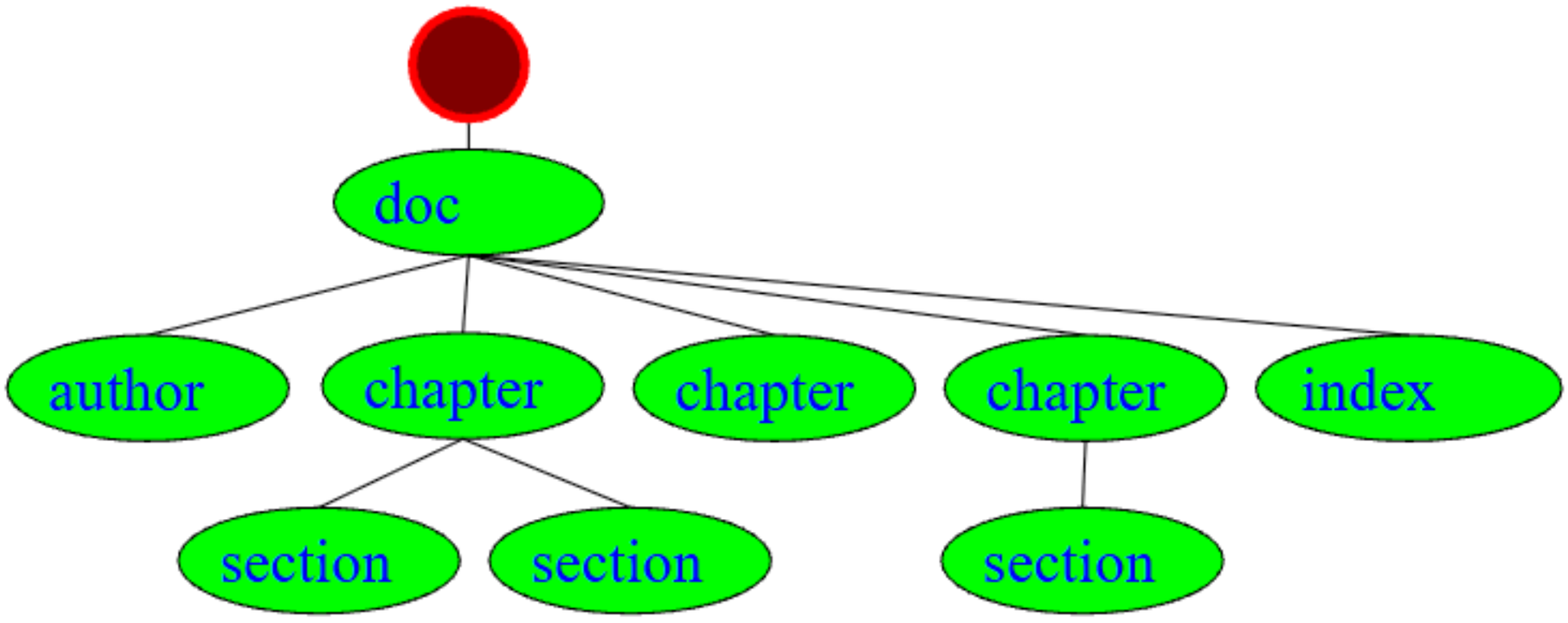
- The Path Expressions can be used to extract values from XML nodes and trees, and to check their properties.
- Recall that, as with any other expression in XQuery, the Path Expressions elaborate **sequences**.
- A Path Expression consists of a series of steps, separated by the character */*.
- Each step is evaluated in a **context**, i.e. a sequence of nodes (with additional information, for example the position of the node), and produces a sequence.
- The next step is evaluated using as context the sequence of nodes produced by the previous step.

Example of evaluation (step 1)

fn:doc(example.xml') / child::doc / child::chapter / child::section

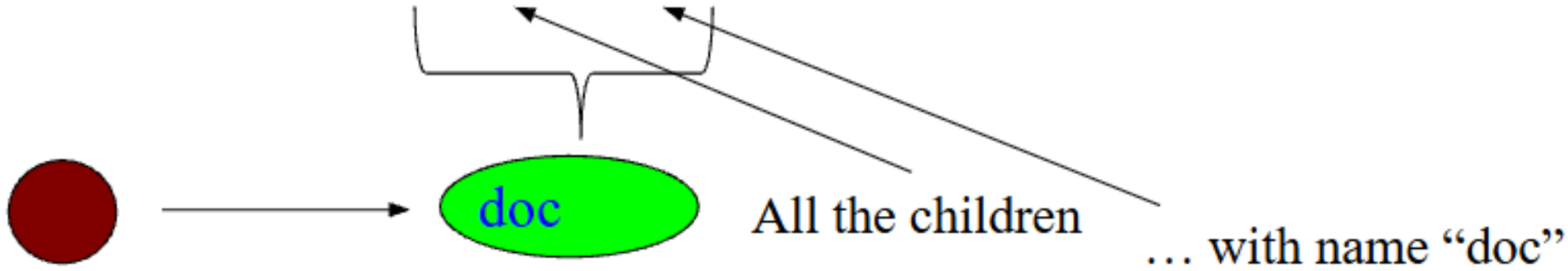


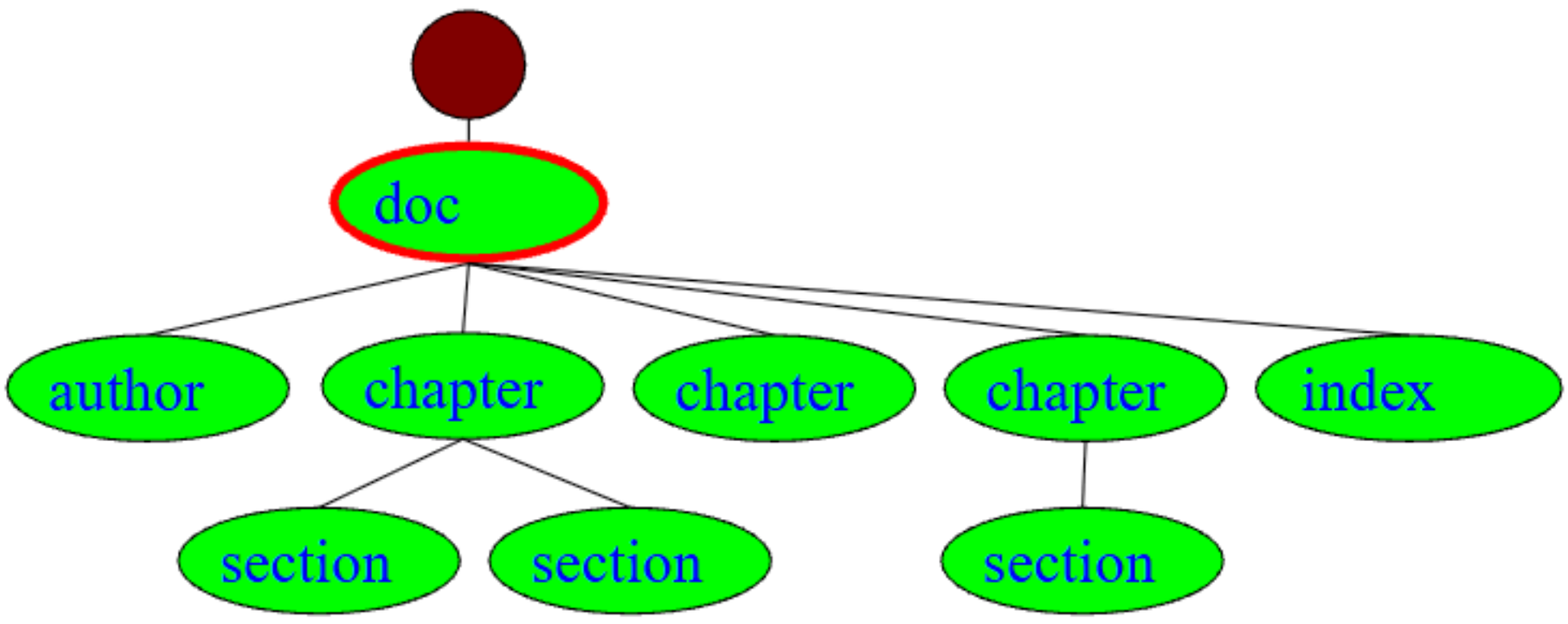
Standard input function



Example of evaluation (step 2)

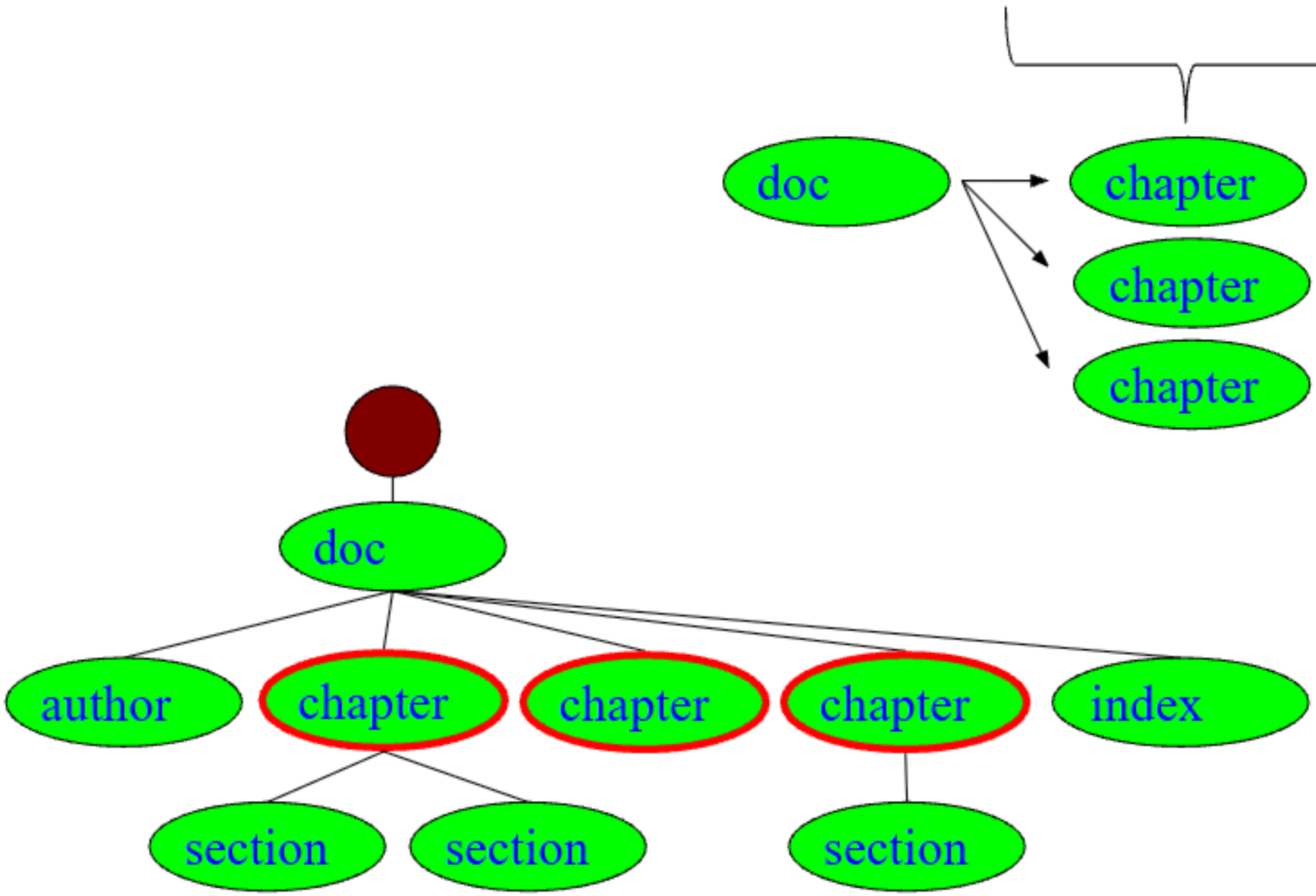
fn:doc(‘example.xml’) / child::doc / child::chapter / child::section





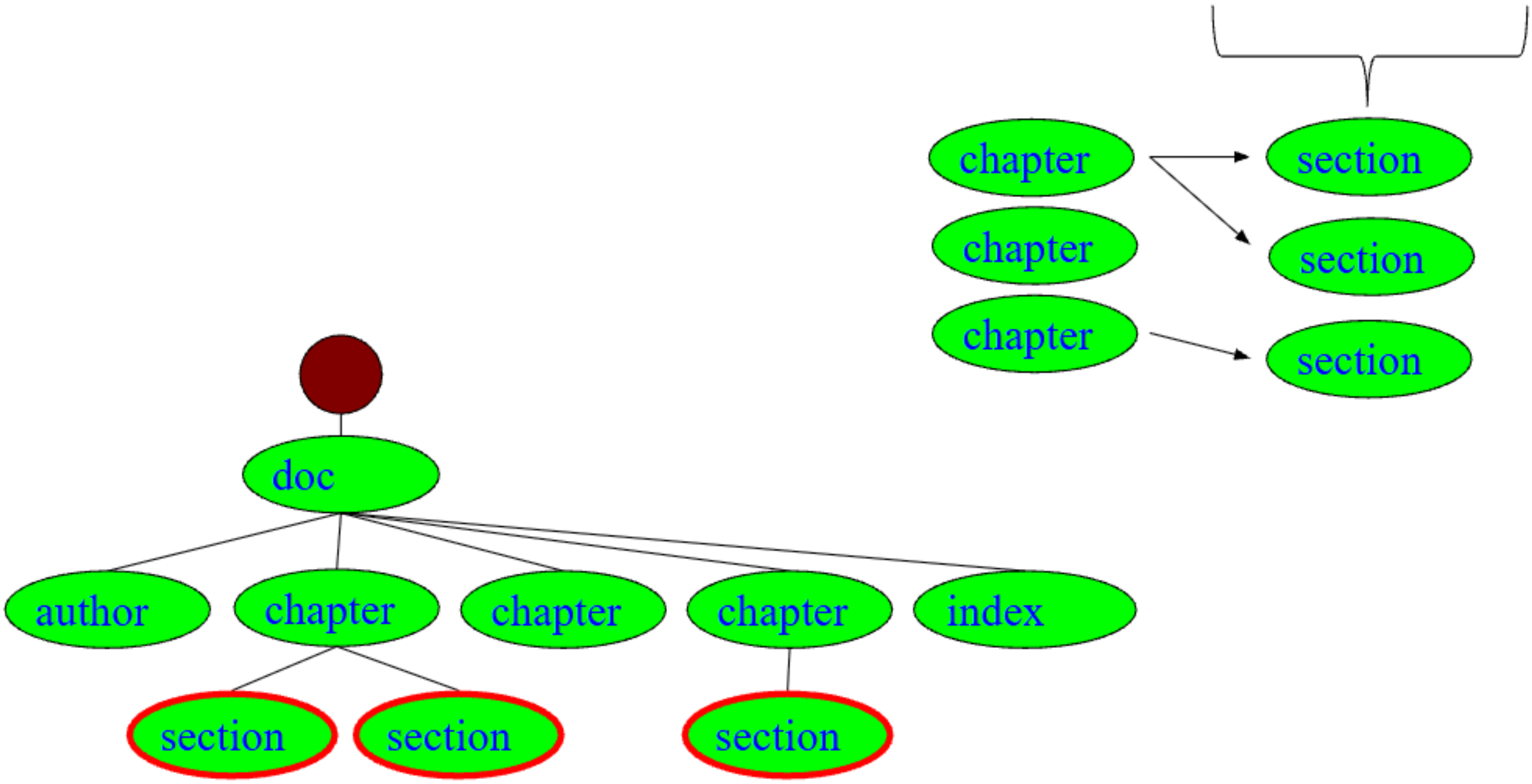
Example of evaluation (step 3)

fn:doc(‘example.xml’) / child::doc / child::chapter / child::section



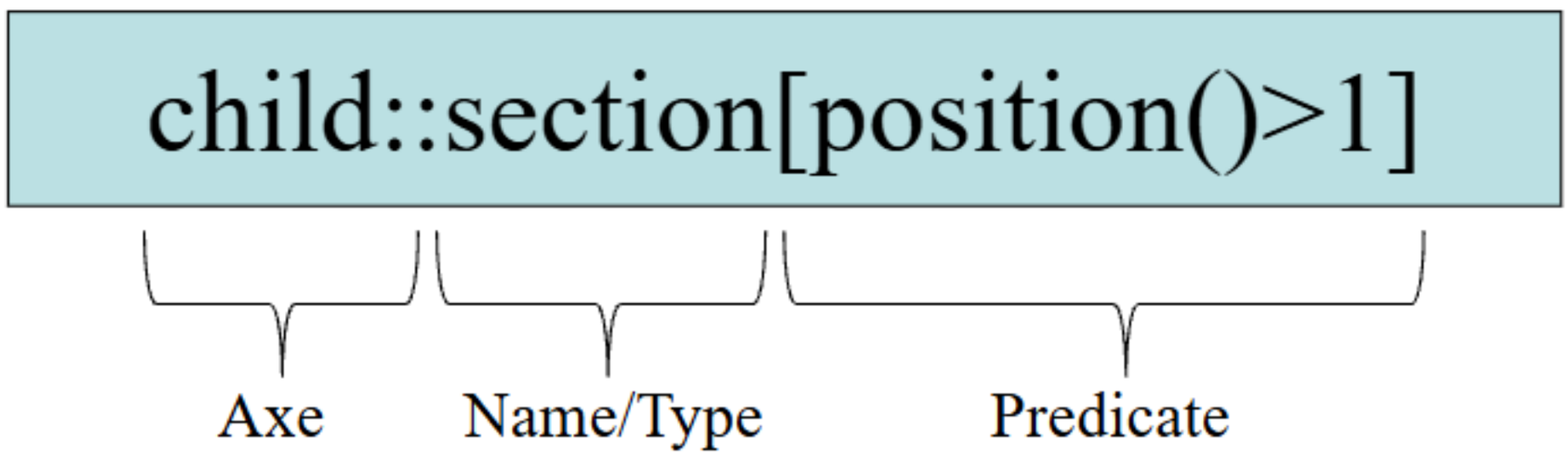
Example of evaluation (step 4)

fn:doc(‘example.xml’) / child::doc / child::chapter / child::section



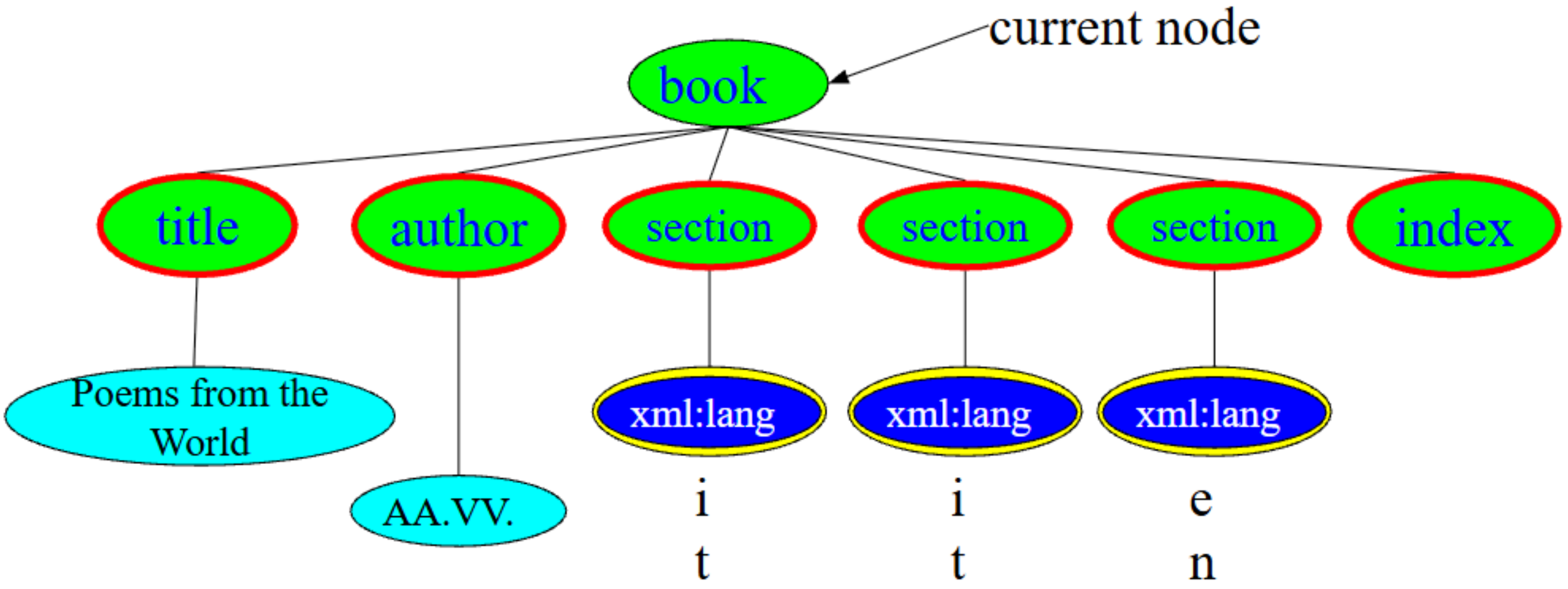
Step structure

- The step of a Path Expression can be made of three main parts:
  - An axe, that select nodes depending on its position w.r.t. the context node (in the example, the children – **child::**).
  - A test that filters these nodes depending on their name and their type (in the example, **section**).
  - One or more predicates, which further filter the nodes depending on more generic criteria (in the example, the fact of **not being the first child**).



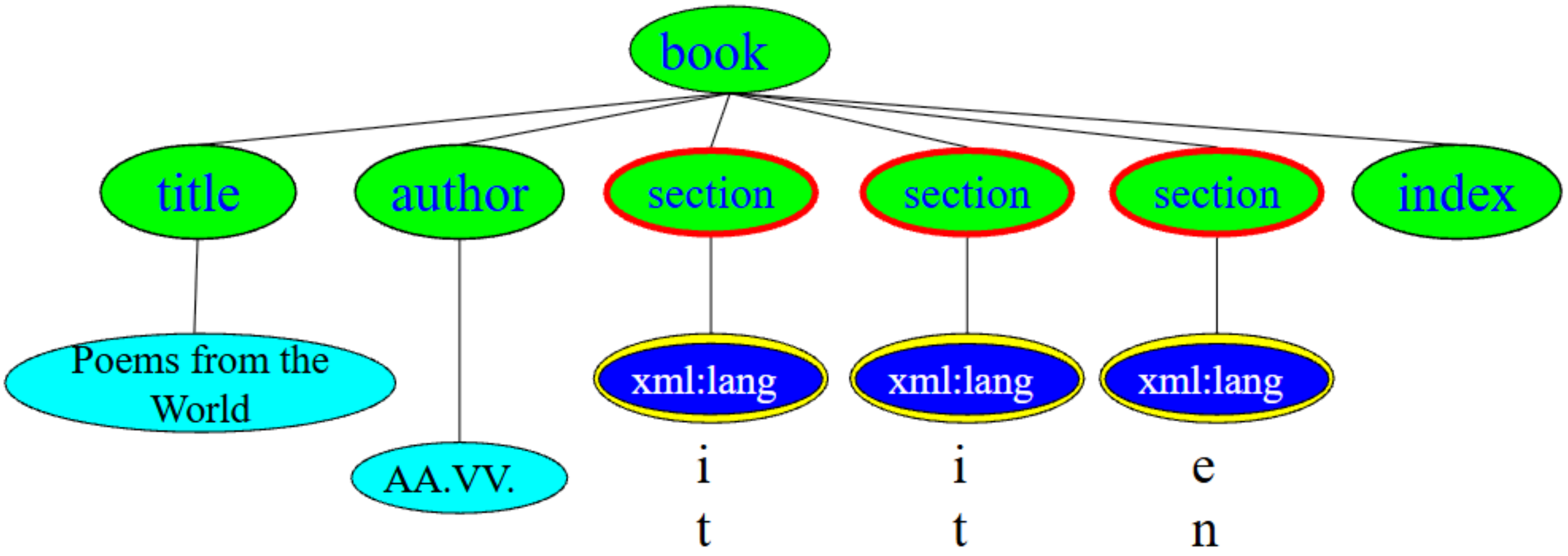
Example of evaluation in one step (1)

child::section[attribute::xml:lang = 'it']



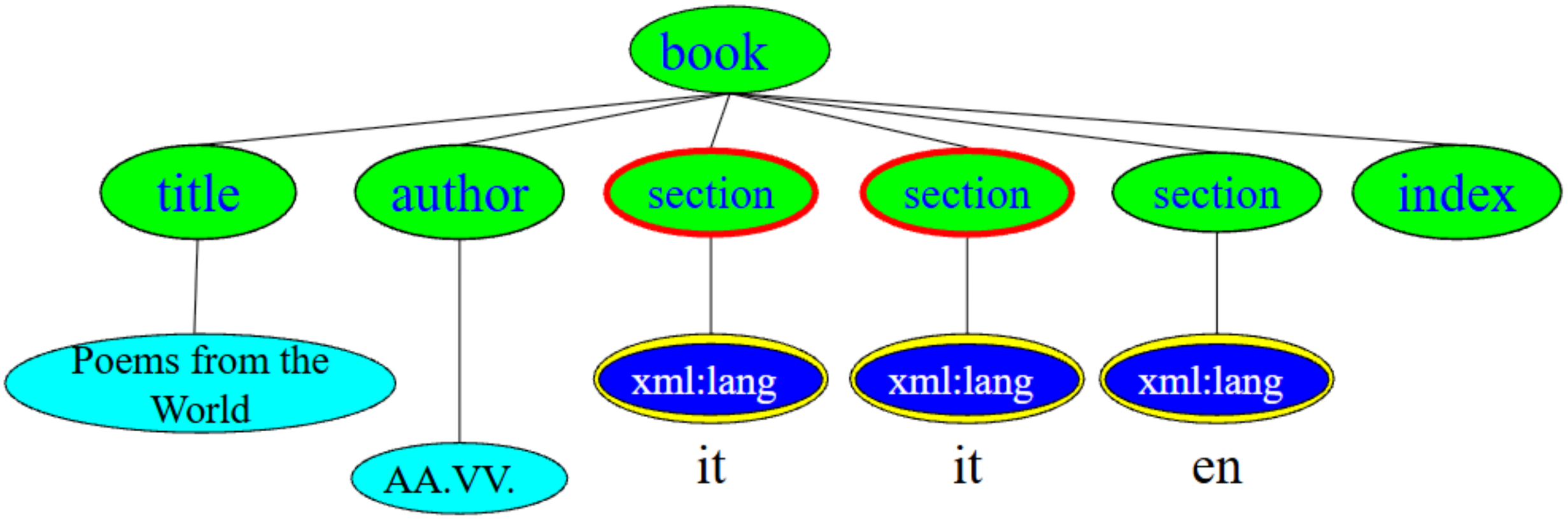
Example of evaluation in one step (2)

child::section[attribute::xml:lang = 'it']



Example of evaluation in one step (3)

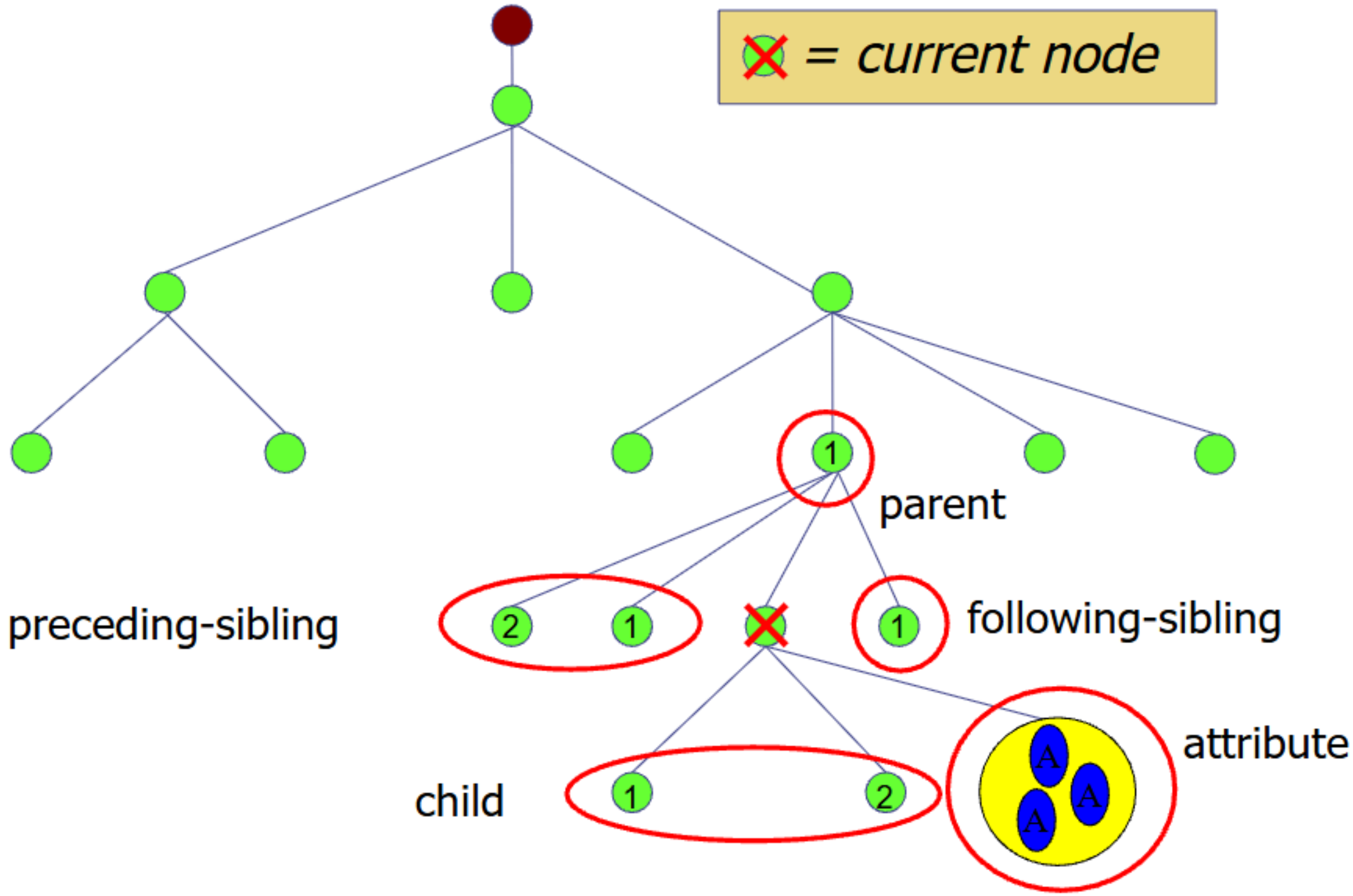
child::section[attribute::xml:lang = 'it']



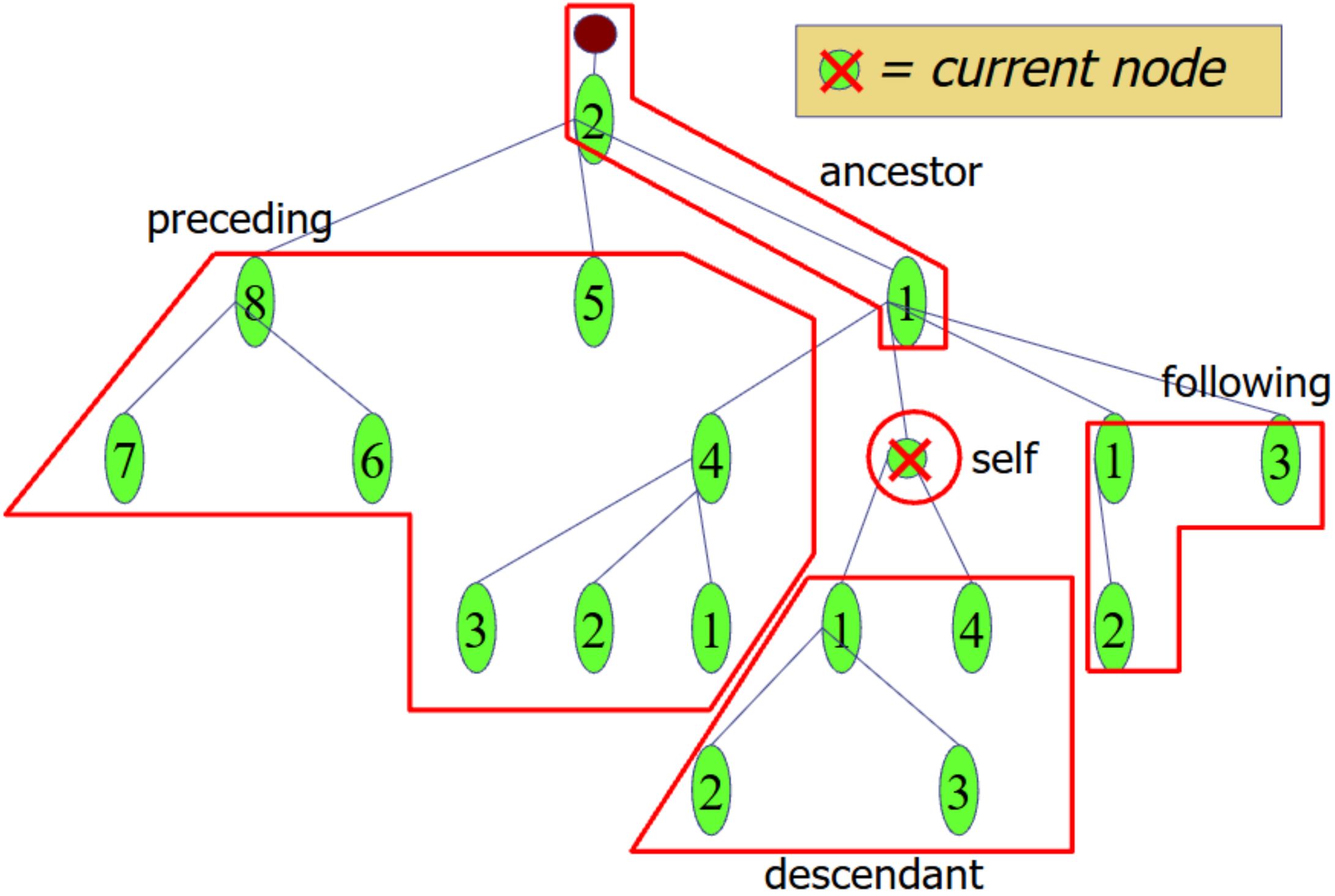
Axes

- The main axes defined in XQuery (and XPath) are the following, exemplified in the next slides:
  - self::
  - child::
  - parent::
  - ancestor::
  - descendant::
  - following-sibling::
  - preceding-sibling::
  - attribute::
- In addition, there are combined axes: descendant-or-self:: and ancestor-or-self::

Example of Axes 1/2



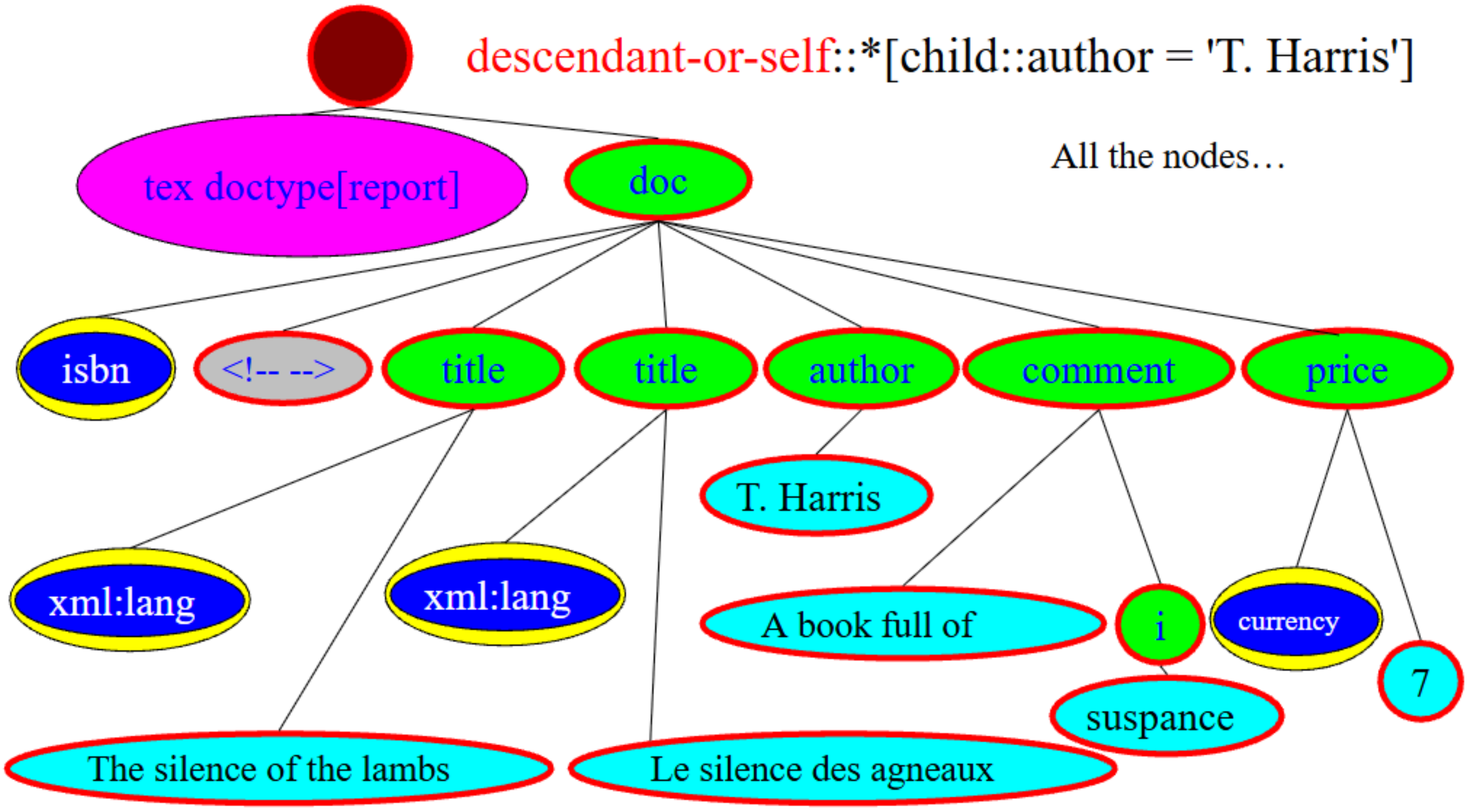
Example of Axes 2/2



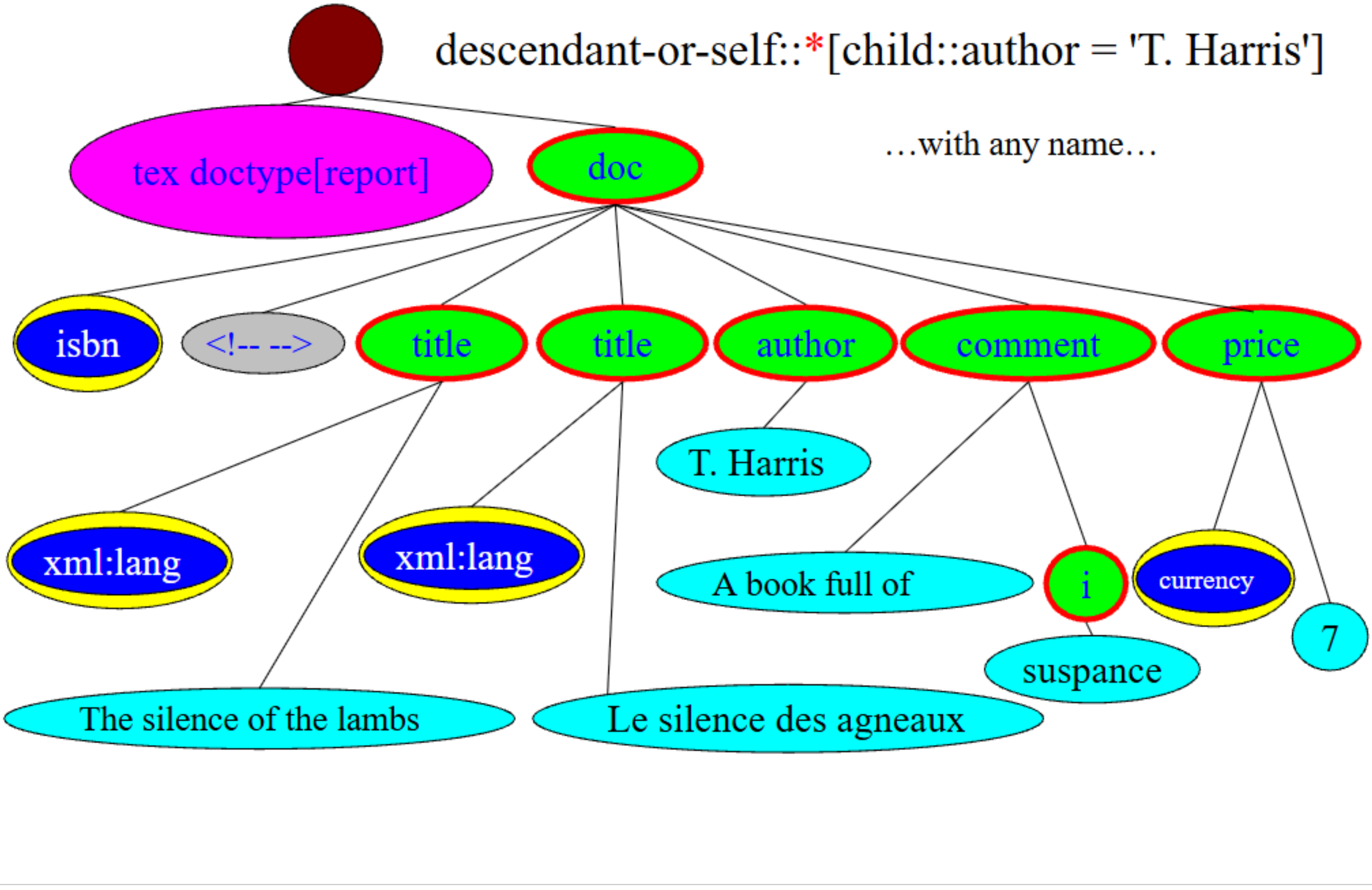
Tests on names/types of node: examples

- The second component of a navigation step, filters the nodes selected by the axe, verifying **the name**:
  - **child::section** returns only the child elements with tag `<section>`.
  - **child::*** returns all the child elements.
  - **attribute::xml:lang** returns the attribute xml:lang.
- Or **the type**:
  - **descendant::node()** returns all the descendants nodes.
  - **descendant::text()** returns all the descendants nodes of type text.
  - **descendant::element()** returns all the descendants elements nodes.
- Or **both**:
  - **descendant::element(person, xs:decimal)** – person elements of decimal type.

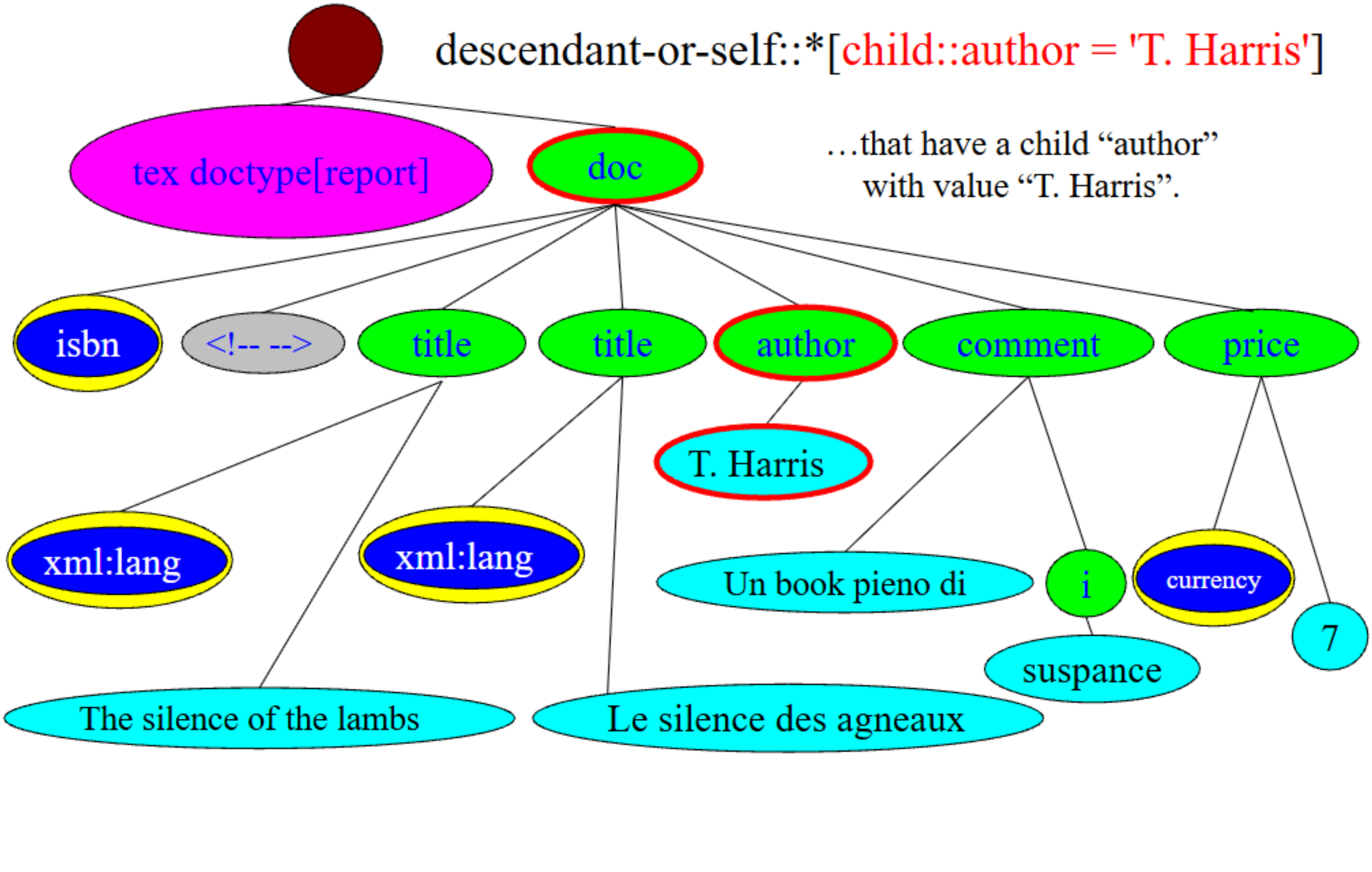
Another example of a step 1/3



Another example of a step (2)

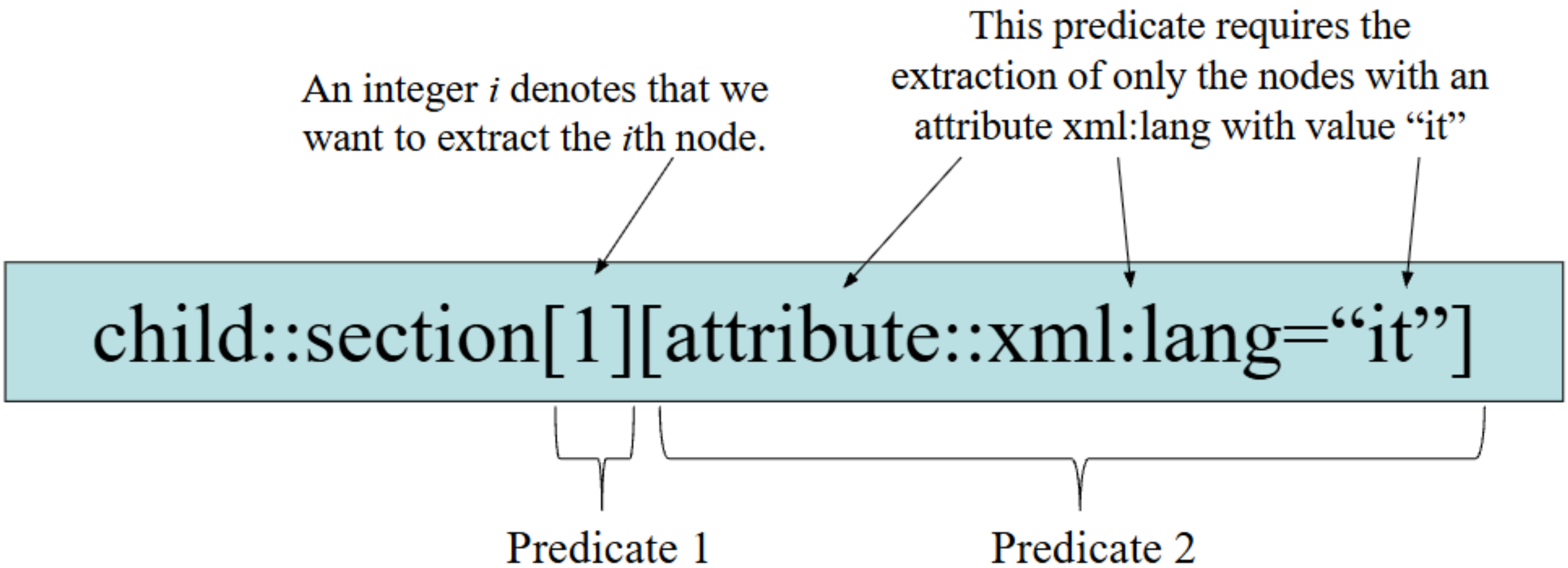


Another example of a step (3)



Predicates

- Every step can end with one or more predicates (in conjunction with *and*), included between "[" and "]", to further filter the nodes selected using axes and tests on names/types.



Evaluation of Predicates

- If the expression returns a single integer value, it has value true if the position of the subject node in the evaluated expression correspond to the value.
  - **child::chapter[2]** returns only the second child with the tag `<chapter>`.
- If the expression returns a non-empty sequence, the predicate is false, conversely if the first item is a node it returns true.
  - **child::chapter[child::title]** returns all the children with tag `<chapter>` that have at least a child with tag `<title>`.
- Otherwise, the standard practices for predicates will be followed:
  - **child::chapter[attribute::xml:lang = “it”]** returns all the children with tag `<chapter>` that have an attribute xml:lang with value “it”.

Complete Path Expressions

- A path expression can also begin with the following prefixes:
  - With the / character: it corresponds to an input sequence of the expression that contains the tree's root.
  - With the // characters: it corresponds to an input sequence of the expression that contains all the nodes in the document.
- It follows some examples of complete path expressions:
  - **/descendant::figure[fn:position() = 42]**

    Select the 42th figure of the document.
  - **/child::book / child::chapter[5] / child::section[2]**

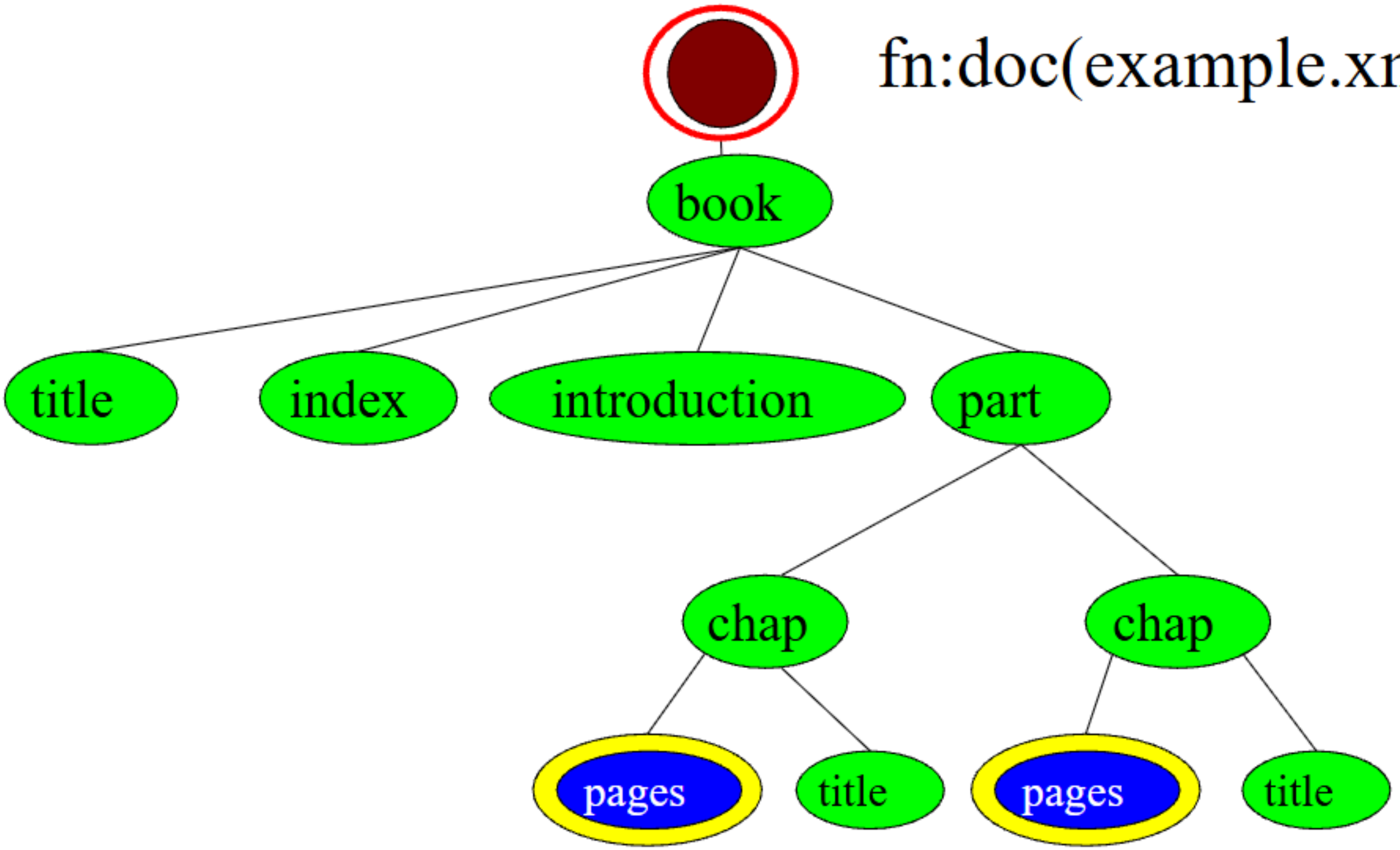
    Select the second section of the fifth chapter.
  - **//self::chapter[child::title]**

    Select all the chapters that have at least one child `<title>`.

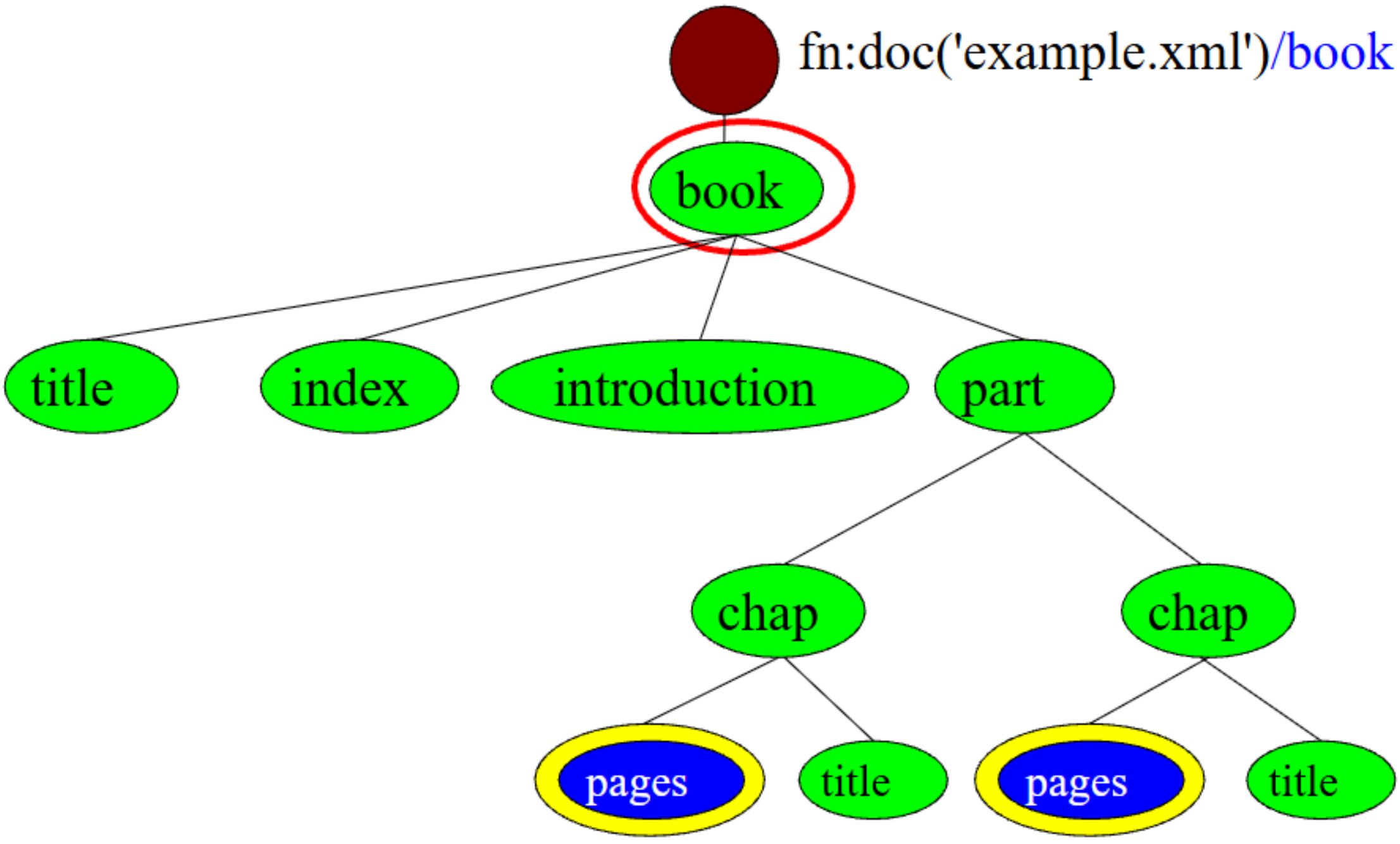
Short syntax

- In order to write more compact expressions, it is possible to use some shortcuts:
  - Omission of the child:: axe, for example:
    child::section/child::paragraph -> section/paragraph
  - Substitution of the axe attribute:: with the character @, for example:
    para[attribute::type="warning"] -> para[@type="warning"]
  - Substitution of descendant-or-self::node() with double slash (//):
    div/descendant-or-self::node()/child::paragraph -> div//paragraph
  - Substitution di self::node() with a dot (.):
    self::node()/descendant-or-self::node()/child::para -> .//para
  - Substitution of parent::node() with two dots (..):
    parent::node()/child::section -> ../section

Examples 1/6

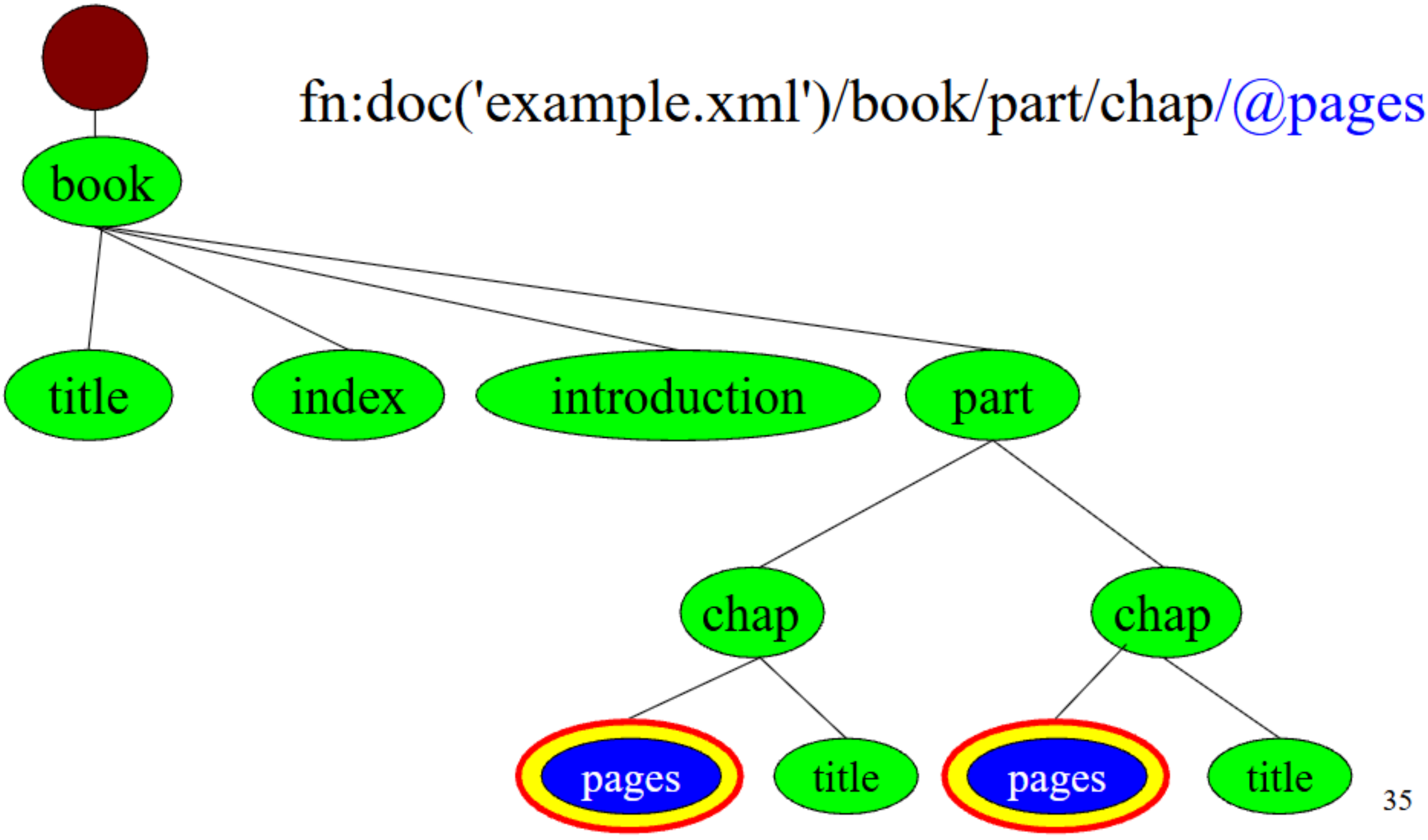

fn:doc(example.xml')
book
title
index
introduction
part
chap
chap
pages
title
pages
title

Examples 2/6


fn:doc('example.xml')/book
book
title
index
introduction
part
chap
chap
pages
title
pages
title

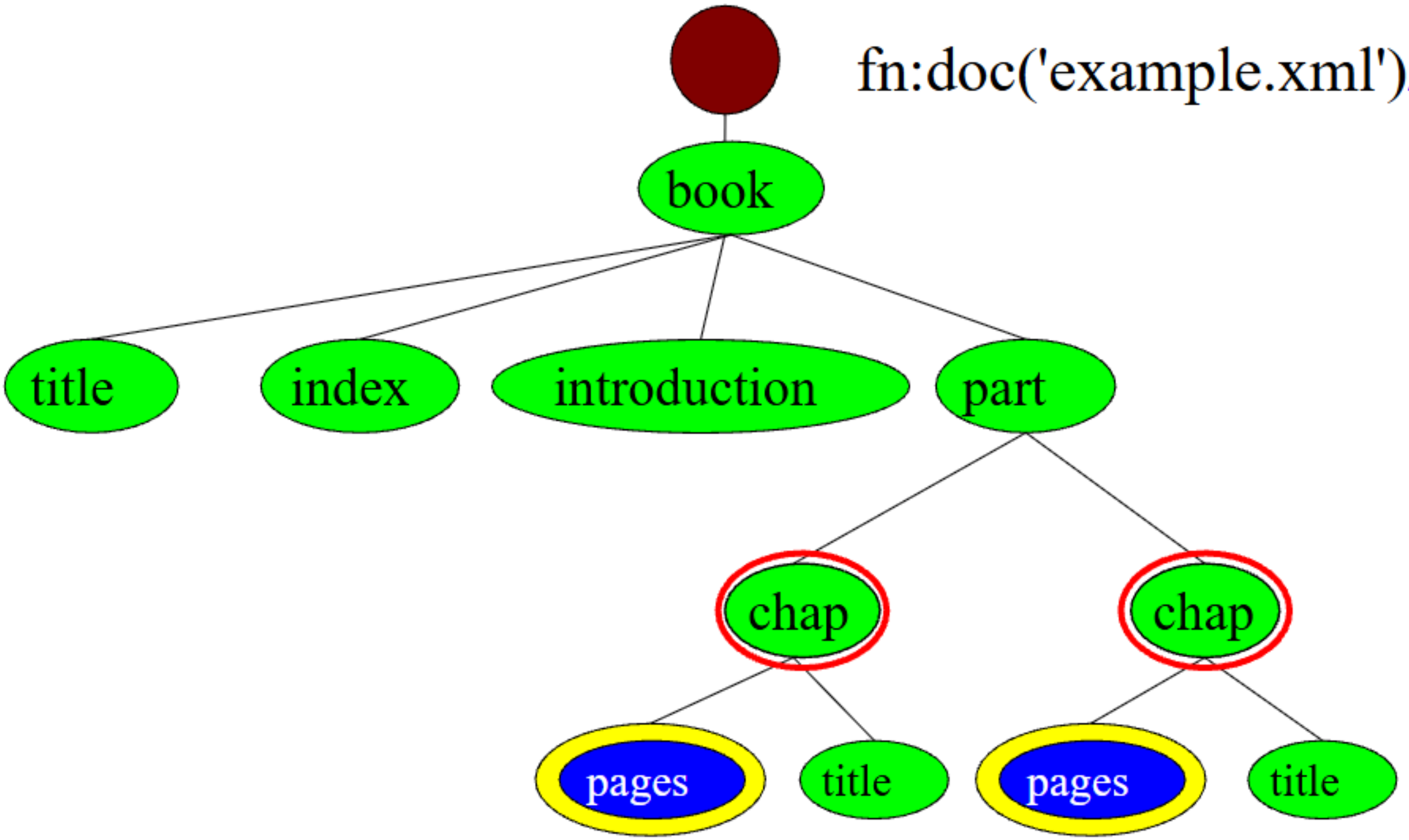
Examples 3/6

fn:doc('example.xml')/book/part/chap/@pages

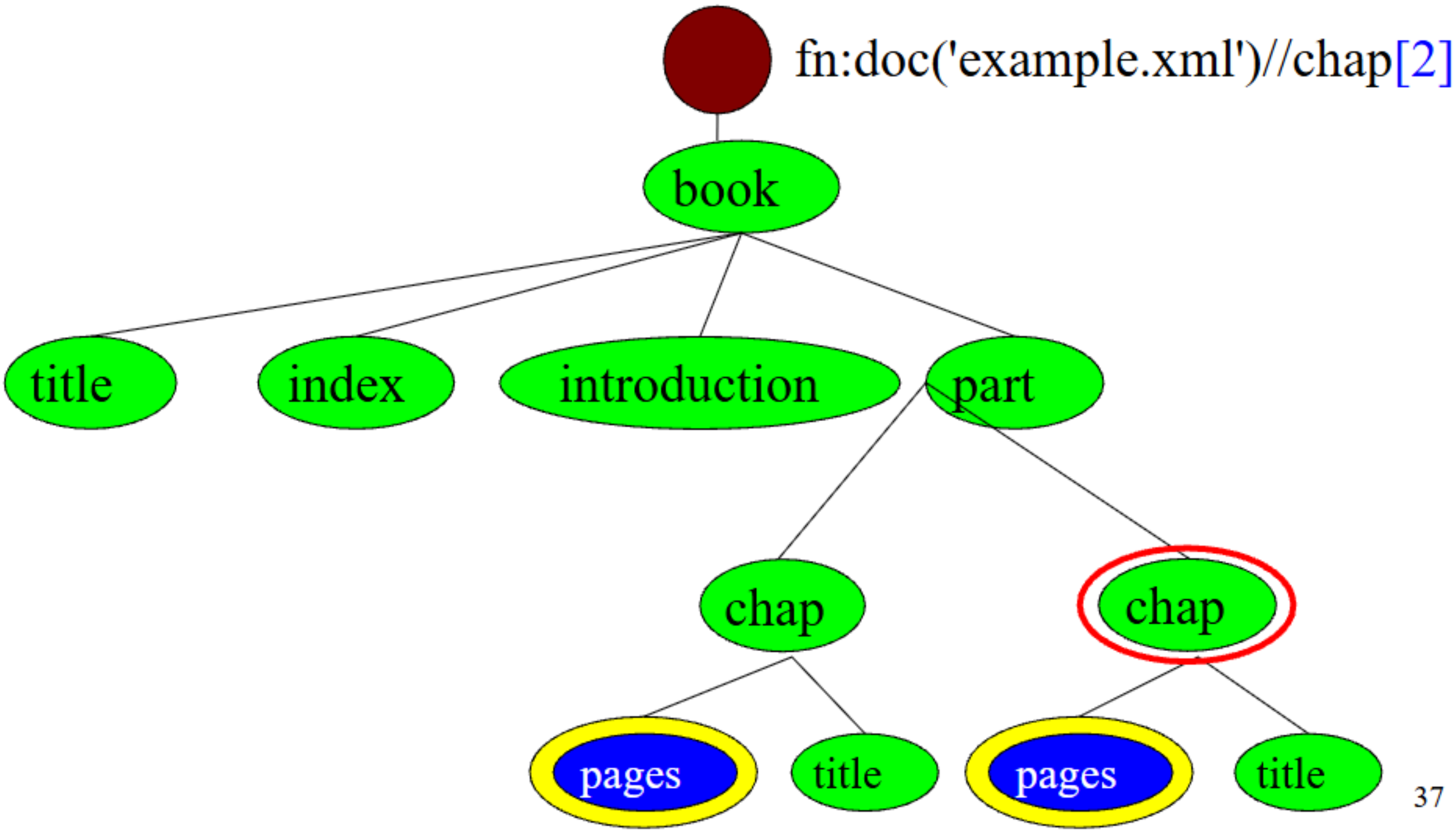


Examples 4/6

fn:doc('example.xml')//chap

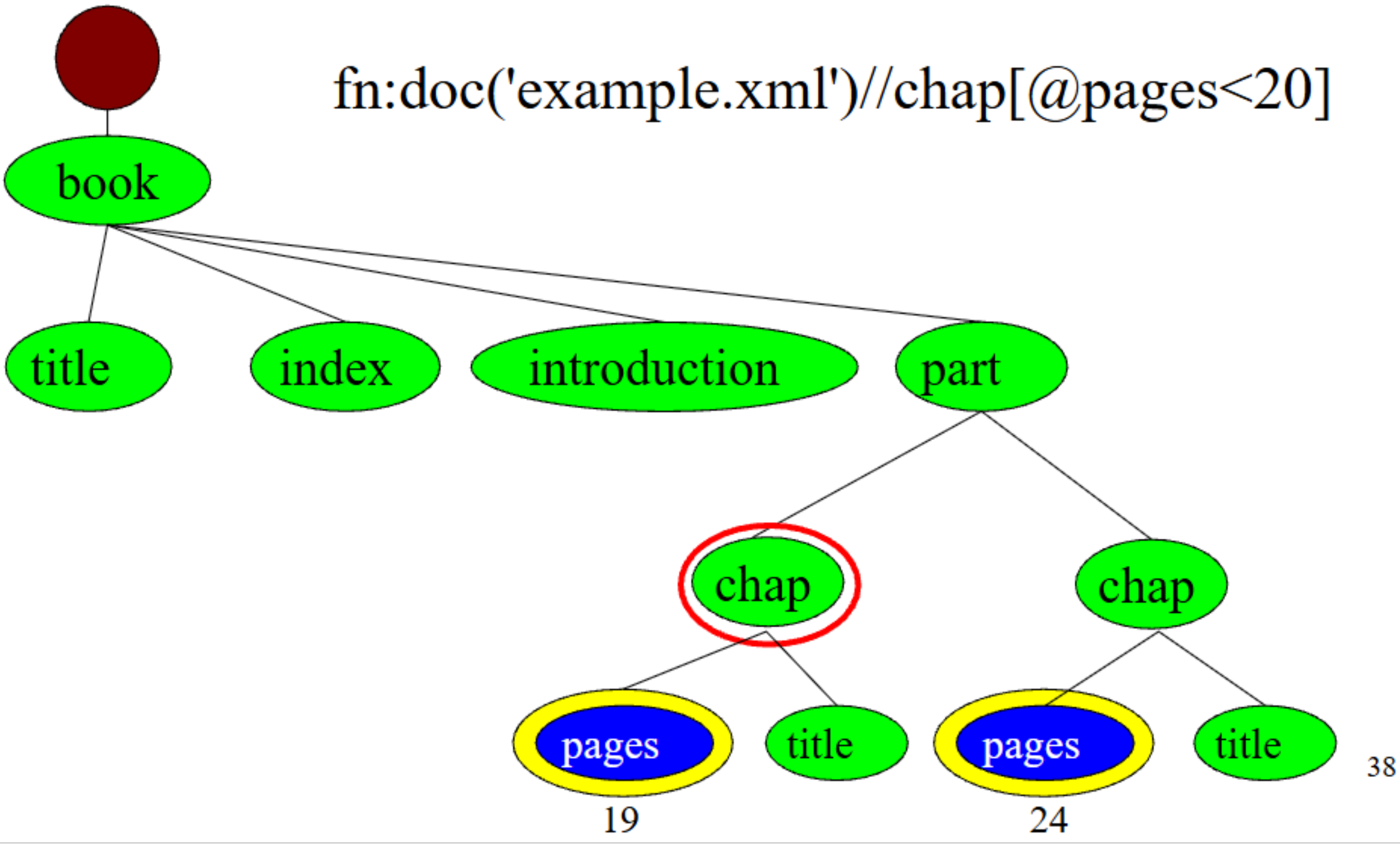


Examples 5/6


fn:doc('example.xml')//chap[2]
book
title
index
introduction
part
chap
chap
pages
title
pages
title

Examples 6/6

fn:doc('example.xml')//chap[@pages<20]



FLWOR Expressions

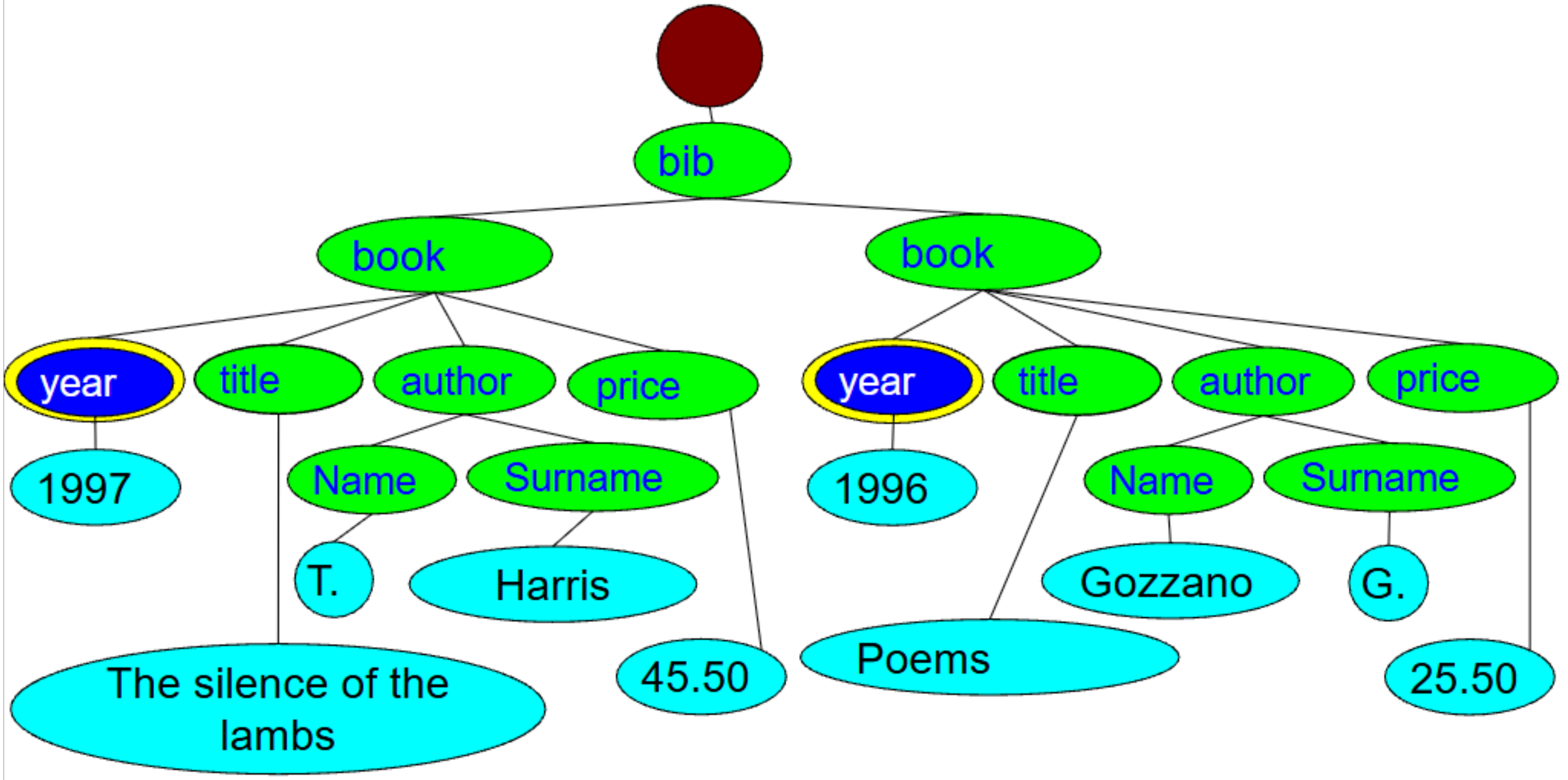
FLWOR Expressions

- A FLWOR expression (pronounced “flower”) is similar to an SQL statement Select-From-Where; it is howevers defined in terms of variables binding. It’s made of 5 parts, some of them are optional:
  - **For**: associate one or more variables to expressions.
  - **Let**: create an alias of the entire result of an expression;
    - For and Let create a list with all the possible associations, also called “tuples” in the XQuery specification.
  - **Where**: filters the list of associations on the basis of a condition;
  - **Order by**: sorts the list of associations;
  - **Return**: create the result of the FLWOR expression.
- These expressions, as any other XQuery expression, can be commented using the symbols (: and :)

  (: This is a comment (: this is a nested comment… :) :)

Iterations of elements – **for** clause 1/5

- For each book, list the year and the title.



Iterations of elements – **for** clause 2/5

- First we select all the books:

  doc("example.xml")/bib/book

- Then **for each book**, that we associate to the variable $b,

  for $b in doc("example.xml")/bib/book

- We write the result:

  return

```
<book year="{Extraction of the year of $b, in XQuery}">
    {Extraction of the title of $b, in XQuery}
</book>
```

Iterations of elements – **for** clause 3/5

```
for $b in doc("example.xml")/bib/book
return
<book year="{ $b/@year }">
    { $b/title }
</book>
```

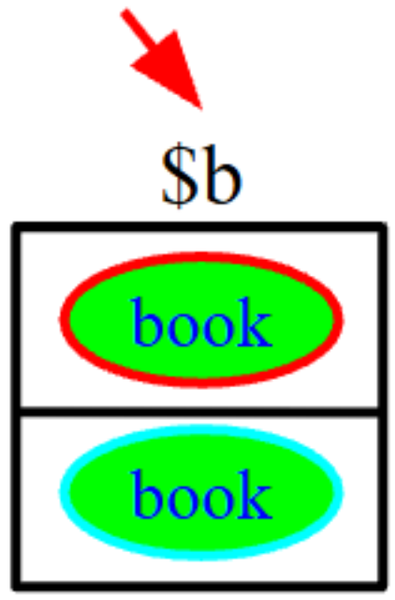
The curly brackets delimit an XQuery expression, which must be evaluated to create the result.

Inside the RETURN clause there can be specified any XQuery expression.
In this case, we are using **constructors** that generates XML code, and are written in XML syntax.

Iterations of elements – **for** clause 4/5

```
for $b in doc("example.xml")/bib/book
return
<book year="{ $b/@year }">
    { $b/title }
</book>
```

Evaluating this expression, we obtain a sequence of nodes (that are associated with the $b variable):



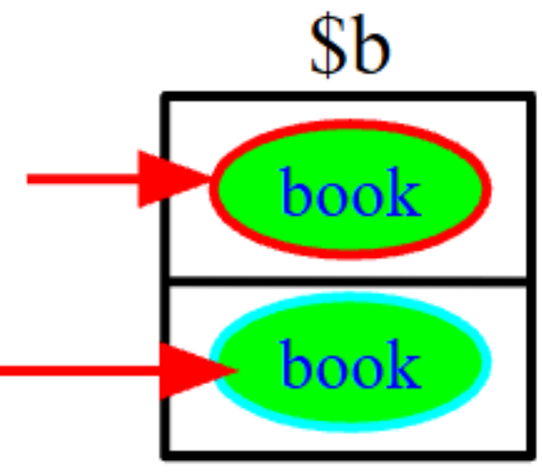
Iterations of elements – **for** clause 5/5

```
for $b in doc("example.xml")/bib/book
return
<book year="{ $b/@year }">
    { $b/title }
</book>
```

For each association ($b) we evaluate this part of the expression:

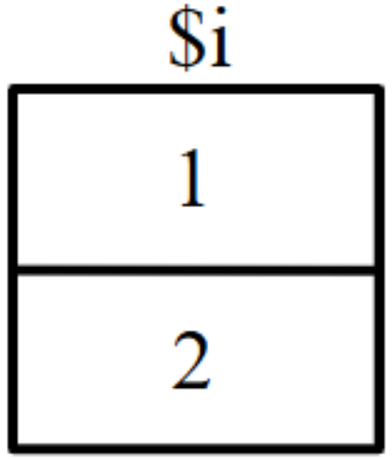
<book year="1997">Dubliners</book>

<book year="1996">Poems</book>



Other examples (1)

- for $i in (1, 2) return $i + 1
- The sequence in input is composed of two elements: 1 and 2.
- For each $i, ($i + 1) is evaluated .
- The result is (2, 3).



Altri esempi (2)

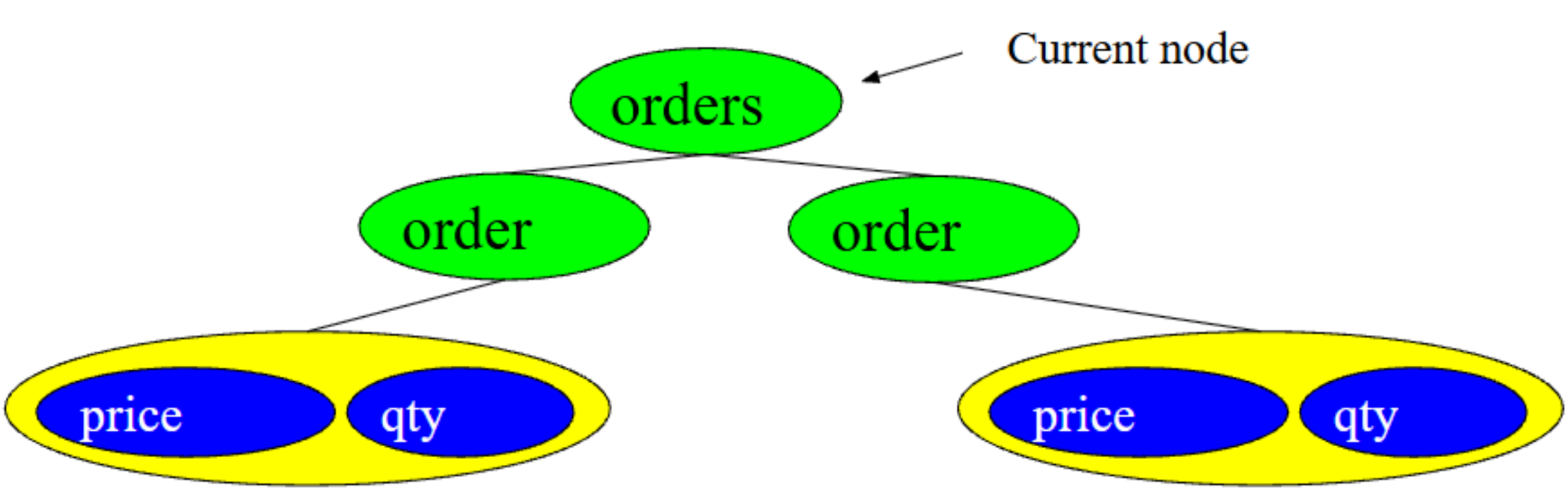
- More than one variable can be used in a single expression.
- for $i in (10, 20), $j in (1, 2) return ($i + $j)
  returns (11, 12, 21, 22)

$i	$j
10	1
10	2
20	1
20	2

The list of associations is given by the cartesian product of all the possible values of the variables

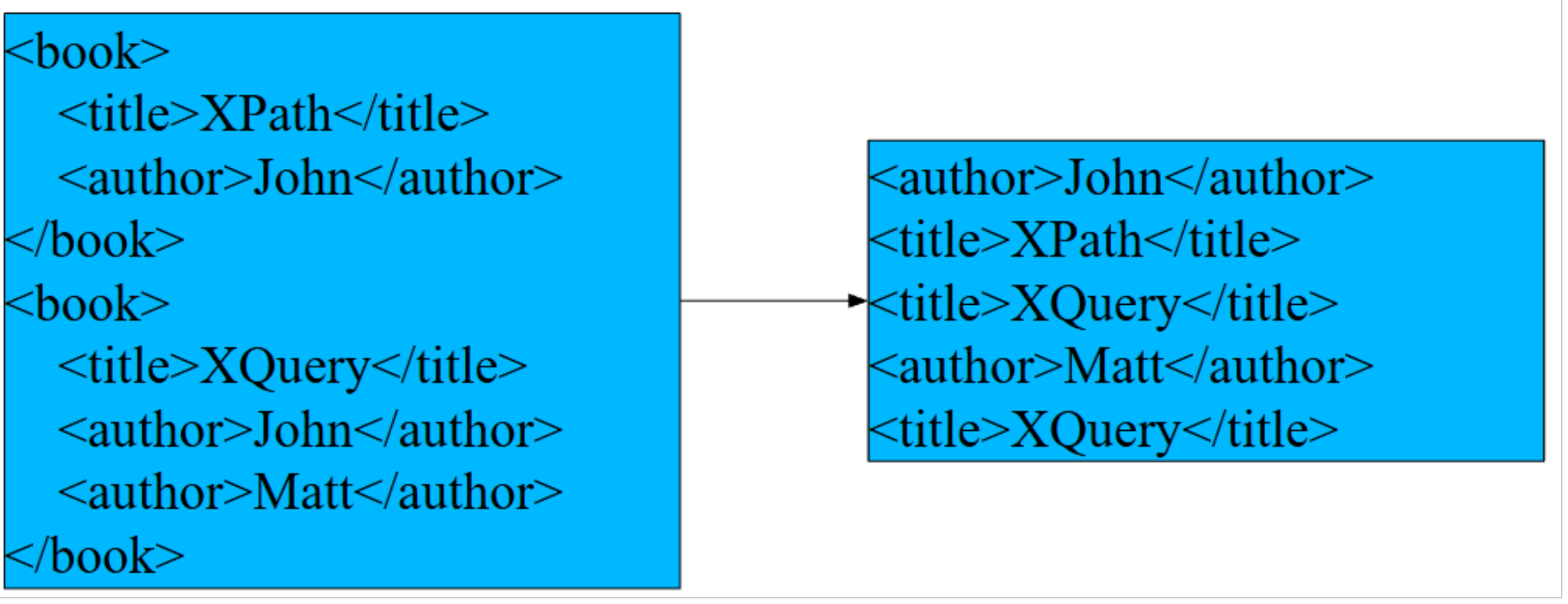
Altri esempi (3)

- Compute the total price of orders:
- sum(for $i in order return $i/@price * $i/@qty)



Altri esempi (4)

- For each author, lists its books (distinct-values is equivalent to the distinct clause of SQL):
- for $a in distinct-values(//author) return
  ($a, for $b in //book[author = $a] return $b/title)



Iterazioni con filtro – clausola where

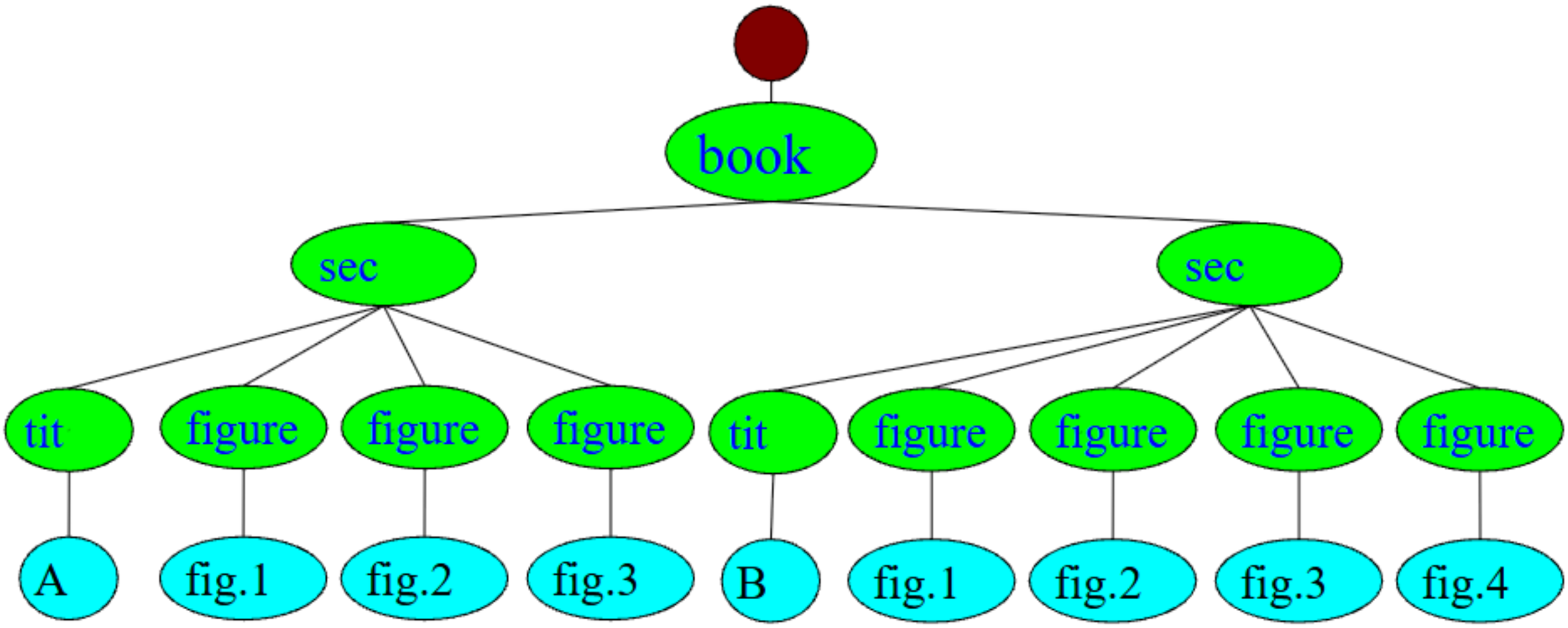
- Find the books published by Anchor Books after the '91.

```
for $b in doc("example.xml")/bib/book
where $b/edizione = "Anchor Books"
        and $b/@anno > 1991
return
<book>
    { $b/title }
</book>
```

With respect to the previous example, we added a filter

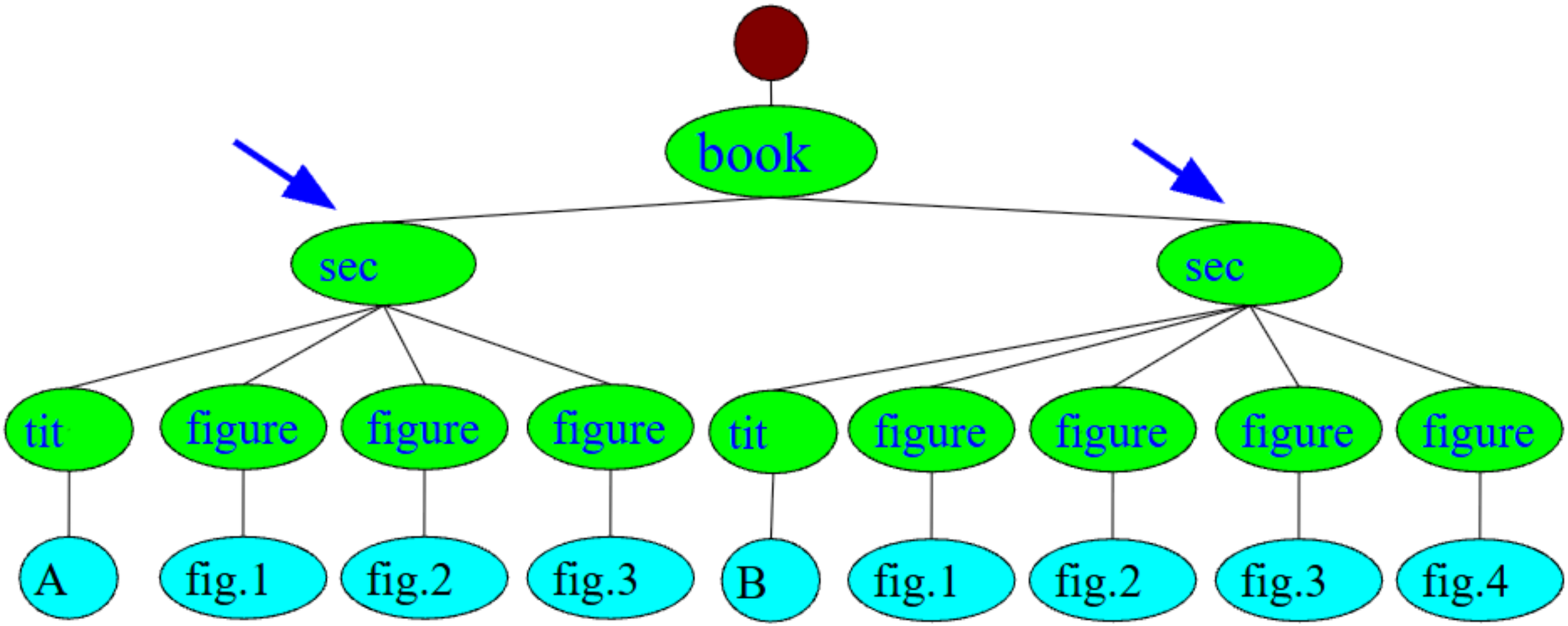
Expressions with aggregated functions – **let** 1/7

- Sometimes its necessary to group elements together.
- This is needed to count them, or to compute their average value, the minimum or the maximum (if we are dealing with numbers).
- For example, we want to list for each section the title and the number of figures



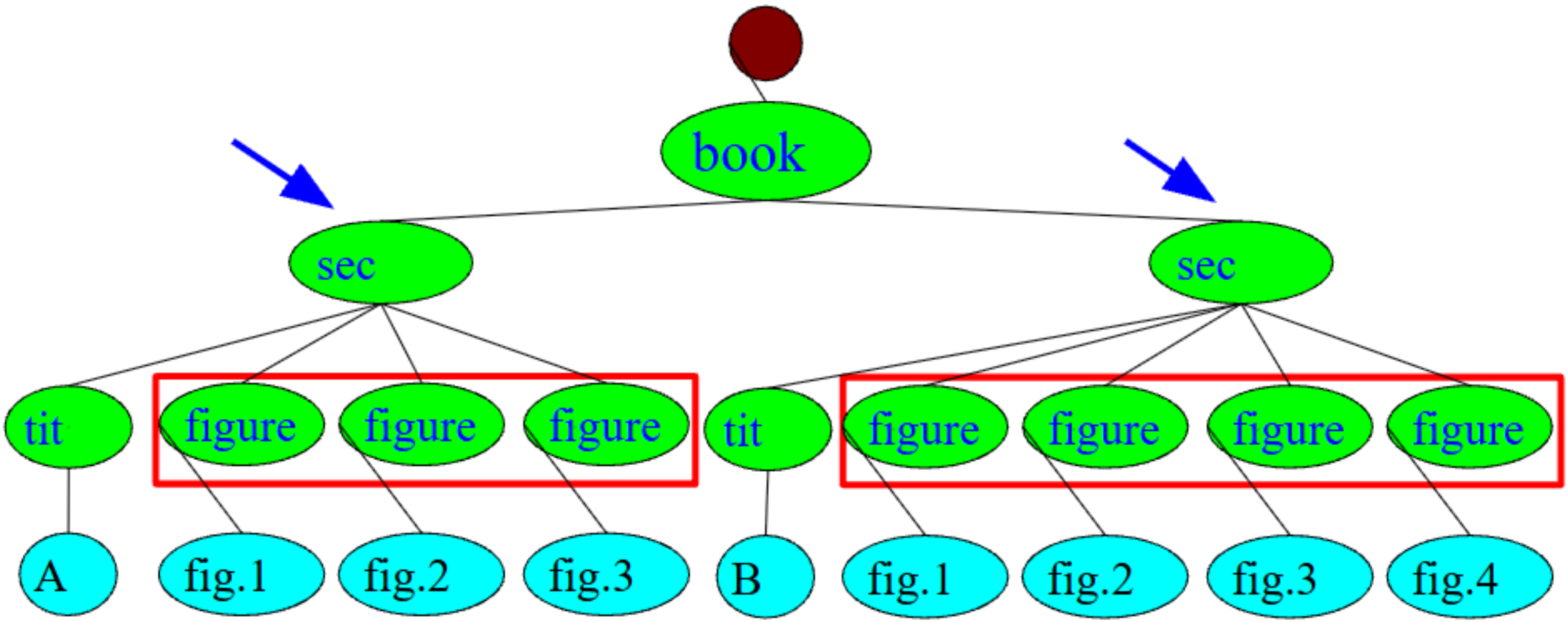
Expressions with aggregated functions – **let** 2/7

- List, for each section, the title and the number of figures.



Expressions with aggregated functions – **let** 3/7

- List, for each section, the title and the number of figures.



Expressions with aggregated functions – **let** 4/7

```
<result>
 {
   for $s in .//section
   let $f := $s/figure
   return
       <section title="{ $s/title/text() }"
       numfig="{ fn:count($f) }"/>
 }
</result>
```

.// select all the descendants of the current node

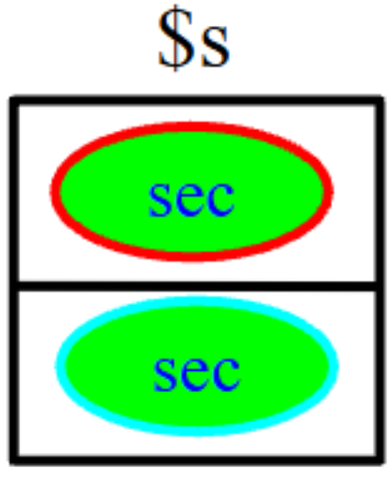
The text() function returns the text value of the node

fn: it's a namespace that denotes that count() is a standard function of XQuery

Expressions with aggregated functions – **let** 5/7

```
<result>
 {
  for $s in .//section
  let $f := $s/figure
  return
      <section title="{ $s/title/text() }"
          numfig="{ count($f) }"/>
 }
</result>
```

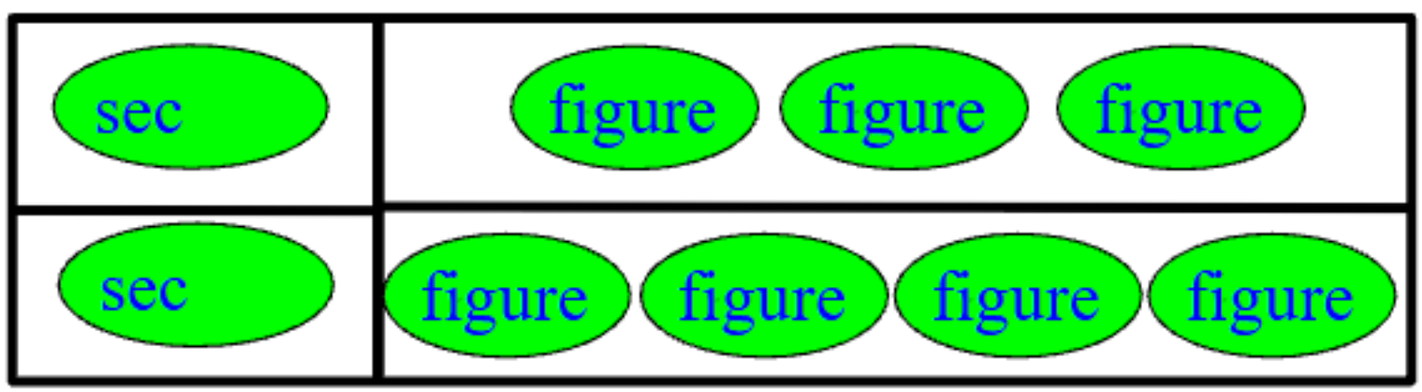
Each node section is associated with $s.



Expressions with aggregated functions – **let** 6/7

```
<result>
 {
   for $s in .//section
   let $f := $s/figure
   return
       <section title="{ $s/title/text() }"
          numfig="{ count($f) }"/>
 }
</result>
```

For each section $s, $f is equal to **the set** of the elements *figure*.

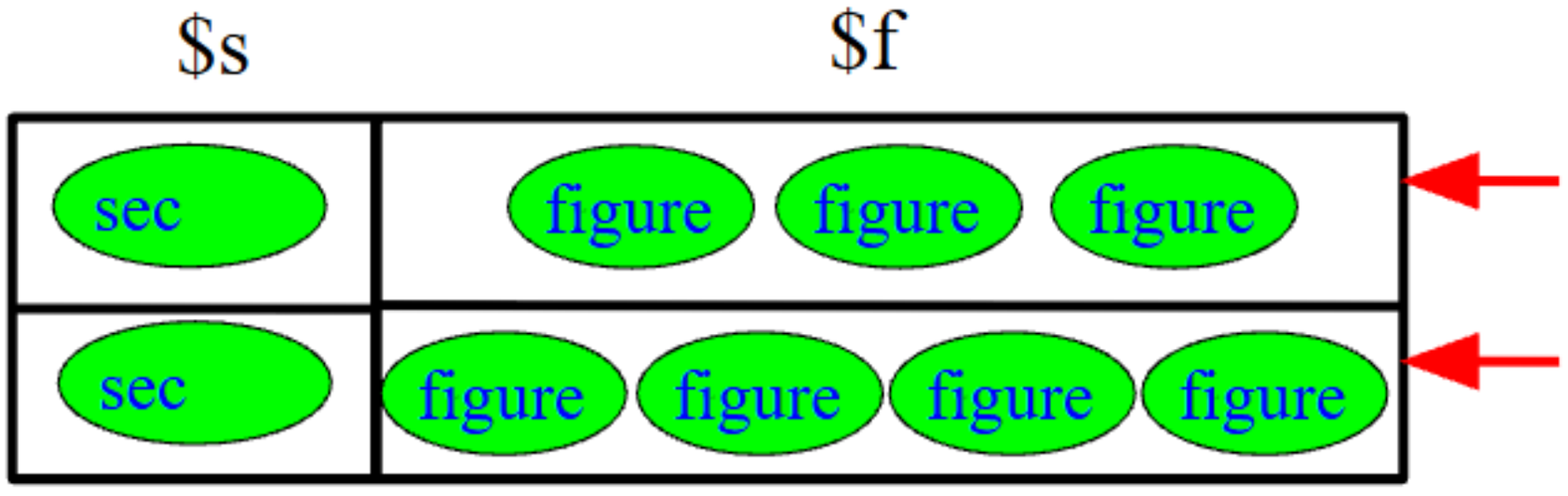


Expressions with aggregated functions – **let** 7/7

```
<result>
 {
   for $s in .//section
   let $f := $s/figure
   return
      <section title="{ $s/title/text() }"
      numfig="{ count($f) }"/>
 }
</result>
```

```
<result>
<section title="A" numfig="3"/>
<section title="B" numfig="4"/>
</result>
```

This part of the expression is evaluated one time for each possible association.

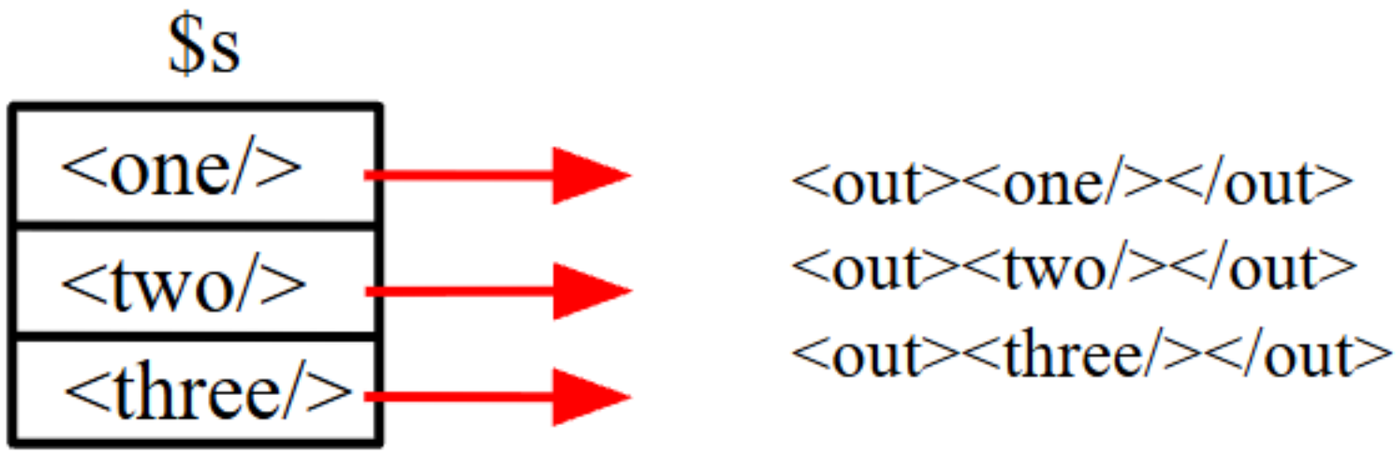


Beware of the difference between FOR and LET

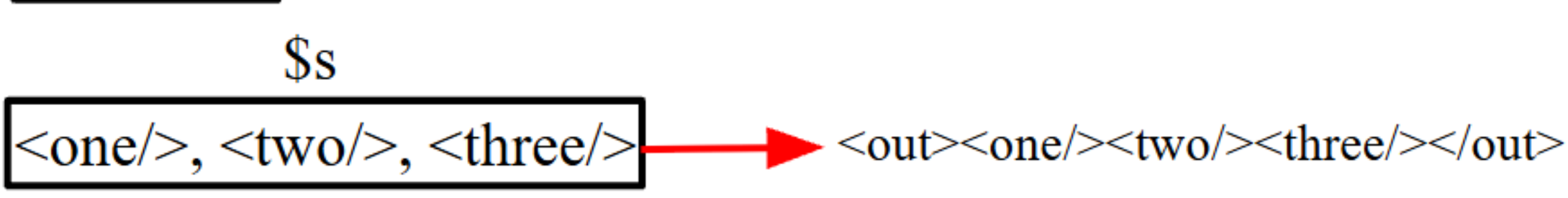
A)
```
for $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

B)
```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

A)



B)



Join in XQuery (1)

- Let's assume having two XML files, one describes the program of a concert, the other describes the composers.
- In both files the composer has a common identifier.
- We want to produce a screening schedule, which lists the songs, with the author and the birth and death dates.

Join in XQuery (2)

conc.xml

```
<concert>
 <song>
    <title>Ciaccona</title>
    <author>B001</author>
 </song>
 <song>
    <title>Polacca</title>
    <author>C001</author>
 </song>
 <song>
    <title>Notturno</title>
    <author>C001</author>
 </song>
</concert>
```

comp.xml

```
<comp>
 <composer>
    <name>J.S.Bach</name>
    <bio>1685-1750</bio>
    <id>B001</id>
 </composer>
 <composer>
    <name>F.Chopin</name>
    <bio>1810-1849</bio>
    <id>C001</id>
 </composer>
</comp>
```

Join in XQuery (3)

```
<program>{
for $comp in fn:doc('comp.xml')/comp/composer,
    $song in fn:doc('conc.xml')/concert/song
where $comp/id eq $song/author
return
  <song>
      <title>{$song/title}</title>
      <di>{$comp/name}</di>
      <date>{$comp/bio}</date>
  </song>
}</program>
```

Join in XQuery (4)

```
<program> {
for $comp in fn:doc('comp.xml')/comp/composer,
    $song in fn:doc('conc.xml')/concert/song
where $comp/id eq $song/author
return
  <song>
    <title>{$song/title}</title>
    <di>{$comp/name}</di>
    <date>{$comp/bio}</date>
  </song>
}</program>
```

$comp	$song
<composer>	<song>
<composer>	<song>
<composer>	<song>
<composer>	<song>
<composer>	<song>
<composer>	<song>

Join in XQuery (5)

```
<program> {
for $comp in doc('comp.xml')/comp/composer,
    $song in doc('conc.xml')/concert/song
where $comp/id eq $song/author
return
  <song>
    <title>{$song/title}</title>
    <di>{$comp/name}</di>
    <date>{$comp/bio}</date>
  </song>
}</program>
```

$comp	$song
<composer>	<song>
~~<composer>~~	~~<song>~~
~~<composer>~~	~~<song>~~
~~<composer>~~	~~<song>~~
<composer>	<song>
<composer>	<song>

Result of the query

```
<program>
  <song>
     <title>Ciaccona</title>
     <of>J.S.Bach</of>
     <date>1685-1750</date>
  </song>
  <song>
     <title>Polacca</title>
     <of>F.Chopin</of>
     <date>1810-1849</date>
  </song>
  ...
```

Sorting – **order by** clause 1/4

- As in SQL, XQuery provides a clause to sort the result of a FLWOR expression.
- ORDER BY can be specified along with several parameters to alter its semantic.
- We will see only the most common use of this clause, through some examples.

Sorting – **order by** clause 2/4

- The employees associated with the variable $employees are returned from the expression, sorted by their salaries.

```
for $i in $employees
order by $i/salary
return $i/surname
```

Sorting – **order by** clause 3/4

- The employees associated with the variable $employees are returned from this expression in order of salary, from the higher to the lower.

```
for $i in $employees
order by $i/salary descending
return $i/surname
```

Sorting – **order by** clause 4/4

- If we want to order the result of a query, which otherwise would be written with a simple navigation expression, we must use a FLWOR expression.

```
for $l in $books//book[price < 50]
order by $l/title
return $l
```

Other expressions

Conditional expressions

- Imperative programming languages (as the C programming language), provide conditional expression in the if-then-else form.
- In XQuery we can also use constructs of this kind.
- This way it is easy to express queries like: add an element <plateNumber> if the status of a car for sale is “registered”.

Conditional expressions: example 1/2

- The evaluation of this expression returns the value of the variable $productx that contains the highest price

```
if ($product1/price < $product2/price)
then $product2
else $product1
```

Conditional expressions: example 2/2

- This query verify the existence of an attribute @discounted, then choose consequently the elements to be selected.

```
if ($product/@discounted)
then $product/wholesale
else $product/retail
```

Comparison operators 1/3

- The following operators act on **sequences** made of several items:
- =, !=, <, <=, >, >=
- For example,

  $book1/author = "Joyce“

  returns true if **at least one of the selected nodes** author has a text value equal to “Joyce”
- These operators must be used with extreme caution, as shown in the following examples.

Comparison operators 2/3

- The fact that = and != are true if **at least one** of the elements of the sequences in comparison satisfies the predicate, produces some counterintuitive behavior:

Ex.1) These operators are not transitive:

- (1, 2) = (2, 3)
- (2, 3) = (3, 4)
- (1, 2) != (3, 4)

Ex.2) Both the following expressions return true:

- (1, 2) = (2, 3)
- (1, 2) != (2, 3)

Comparison operators 3/3

- XPath 2.0 introduces new operators to compare sequences made of **a single** item:
- eq, ne, lt, le, gt, ge
- $book1/author eq "Joyce" is true only if a **single author node** has been selected.
- Otherwise, an error is reported.

Logical and arithmetic operators

- XQuery provides the following logical operators:
- or, and
- A standard function for logical negation:
- fn:not()
- And the following arithmetic operators:
- +, -, * div, idiv (integer division), mod (remainder of the division).
- For example:
- 1 eq 1 and 1.5 eq 3 div 2
- -1 eq -3 idiv 2 or 2 eq 3

Expressions with quantifiers 1/3

- In XQuery, a variable can be associated with a single value. For instance, $price can have an integer value of 12000.
- However, often the variables are associated to sets of objects.
- For example, the expression

  for $lib in doc(books.xml)/books/book

  could associate the variable $lib to several <book> elements.
- Specific operators are needed to verify the properties of sets of objects.

Expressions with quantifiers 2/3

- Let's introduce two operators of XQuery that serve this purpose, with some examples:

  some $emp in //employee

  satisfies ($emp/salary > 13000)

- This expression is true if **at least one** employee receives a salary greater than 13000.

Expressions with quantifiers 3/3

- The opposite result is obtained using the other quantifier: **every**.

  every $imp in //impiegato

  satisfies ($imp/stipendio > 13000)

- This expression is true if **all the** employees receive a salary greater than 13000.

Expressions with quantifiers: examples

```
some $x in (1, 2, 3), $y in (2, 3, 4)
satisfies $x + $y = 4
```

Returns true.

```
every $x in (1, 2, 3), $y in (2, 3, 4)
satisfies $x + $y = 4
```

Returns false.

Some standard functions

Input functions

- They are needed in order to obtain XML code to be queried.
- They are used to access a single document (doc) or a sequence of documents (collection), provided by the manager system.
- fn:doc('bib.xml')
- fn:collection('composers')

Functions on sequences of nodes

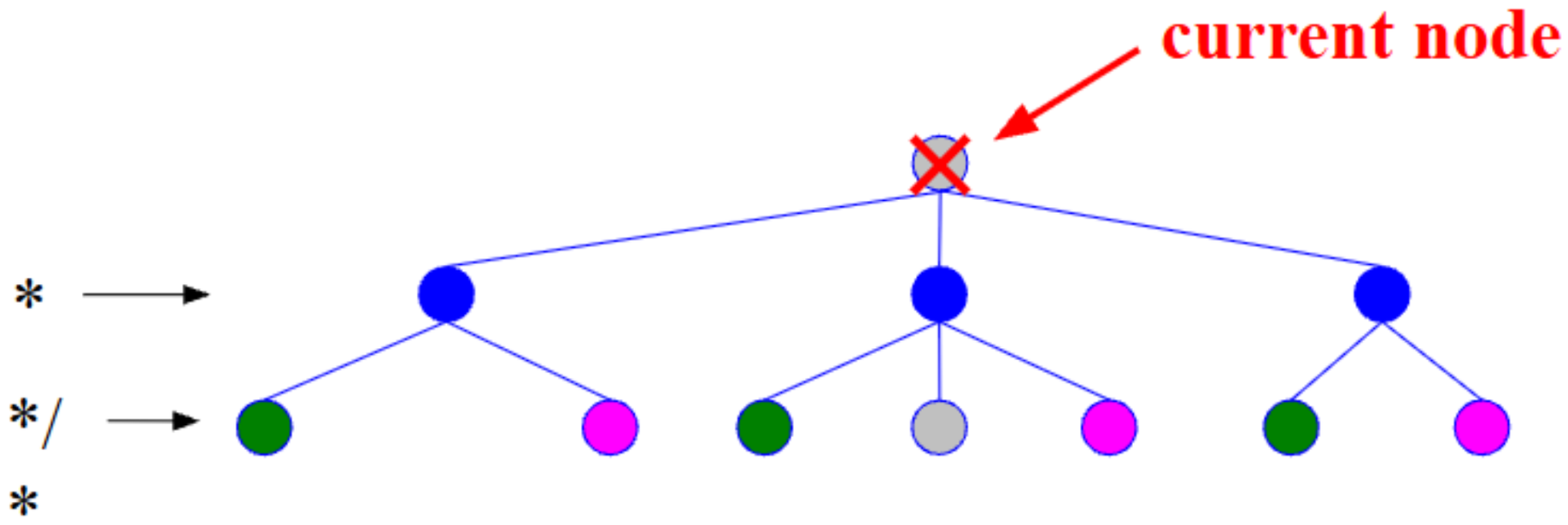
- fn:position()
- Position of current node.

- fn:last()
- Number of nodes.

- fn:count($elements)
- Cardinality of the sequence of nodes in the argument ($elements)

Example

- Each color shows the nodes selected using the corresponding expression:
- fn:count(*) *returns* 3
- */*[fn:position()=1]
- */*[fn:position()=fn:last()]



Aggregated functions

- The following functions are
- The following functions are usually used together with the let construct, as we have already seen.
- count
- avg
- max
- min
- sum

Examples of aggregated functions

- $seq3 = (3, 4, 5)
- count($seq3) returns 3
- avg($seq3) returns 4
- max($seq3) returns 5
- min($seq3) returns 3
- sum($seq3) returns 12

References

- XQuery specifications:
  - http://www.w3.org/XML/Query/
- DBMS supporting XQuery:
  - Oracle database server.
  - DB2.
  - SQL Server.
- List of all the main implementations of XQuery:
  - http://www.w3.org/XML/Query/#implementations