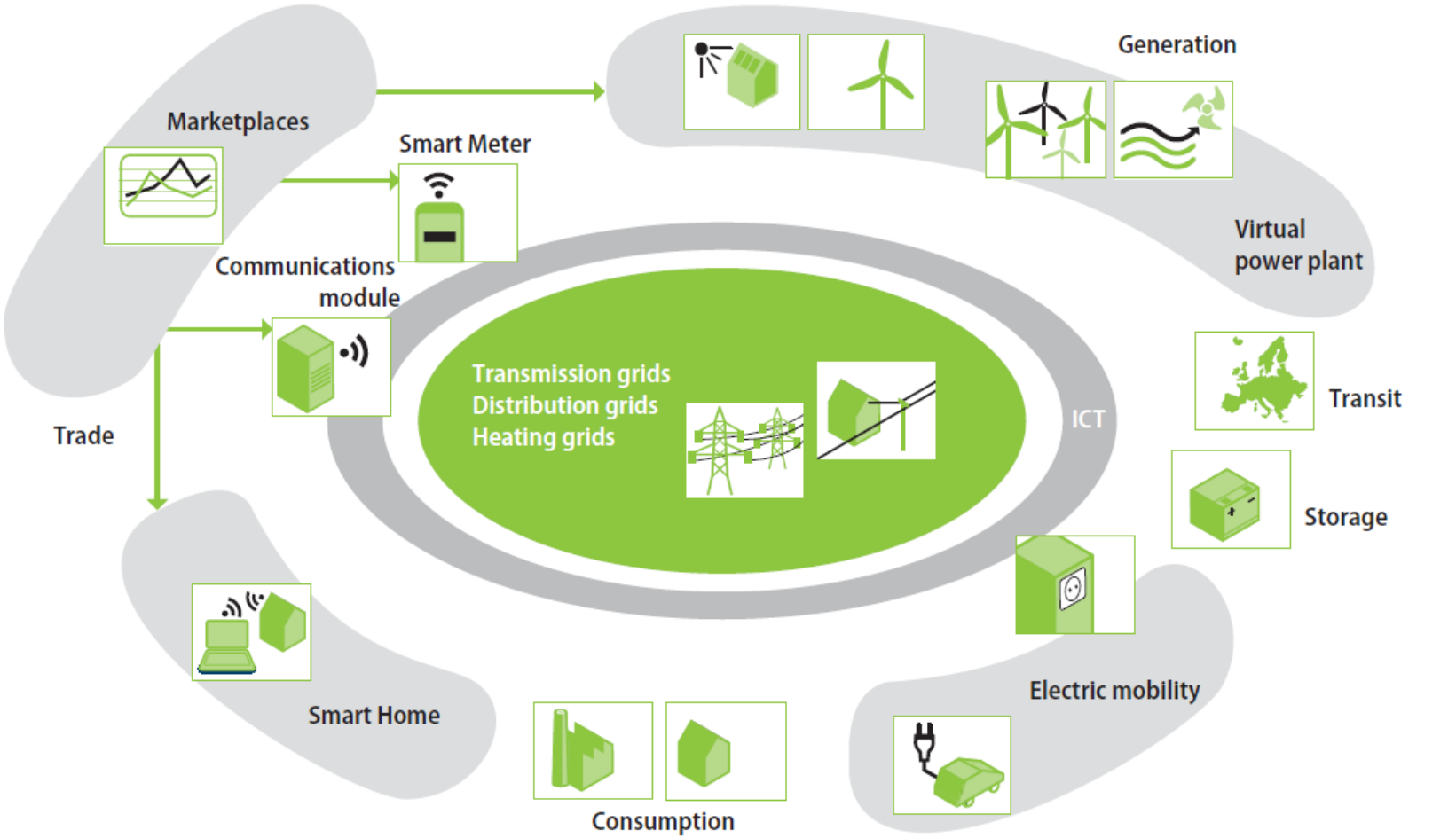
1 Semistructured Data Models

Lots of data is published on the Web

- data is interlinked,
- typically automatically generated from databases,
- using formats making the data self-describing,
- and interchanged between different sites.

Example: Smart Grid



Quelle: acatech, Future Energy Grid

. . . a flexible, however standardized data representation technology is needed.

. . . switching from **structured** relational data model to **semistructured** data models.

- A language to structure data in a flexible way would be helpful.
- A language to semantically annotate data would be helpful.
- This should be done in a way sensible for humans and machines.

1.1. Extensible Markup Language (XML)

Example Document in XML

```
<Order PONumber="1600">
  <Reference>ABULL-20140421</Reference>
  <Requestor>Alexis Bull</Requestor>
  <CostCenter>A50</CostCenter>
  <ShippingInstructions>
     <name>Alexis Bull</name>
     <Address>
       <street>200 Sporting Green</street>
       <city>South San Francisco</city>
       <state>CA</state>
       <zipCode>99236</zipcode>
       <country>United States of America</country>
     </Address>
     <Phone> ... </Phone>
  </ShippingInstructions>
  <SpecialInstructions/>
  <AllowPartialShipment>false</AllowPartialShipment>
  <LineItems>
    ...
  </LineItems>
</Order>
```

XML Basics

- XML is a *Markup Language*; XML is a subset of SGML, which is a metalanguage standardized 1986 used to define the structure and content of documents. (Note: HTML is a specific application of SGML.)

  Markup is indicated by a *start-tag* <aTagname> followed by an *end-tag* </aTagname>, both enclosing the markuped part of the document.

- Start- and corrsponding end-tag, including the enclosed part of the document, is called *element*; the name of the element is the name of the tag and the *content* of the element is the enclosed part.

```
<ShippingInstructions>
    <name>Alexis Bull</name>
    <Address>
        <street>200 Sporting Green</street>
        <city>South San Francisco</city>
        <state>CA</state>
        <zipCode>99236</zipcode>
        <country>United States of America</country>
    </Address>
</ShippingInstructions>
```

. . . continued

- The content of an element may contain simply text without any further tags, only (other) elements, or a mixture of both.
  If the content of an element does not contain any further tags, the content is called *element text*.
  If the content of an element is built out of other elements only, it is called *element content*.
  In the remaining case it is called *mixed content*.
- A tag without content is called *empty tag* with shorthand notation `<aTagname/>`
- Start-tags may be further described using attributes.

Attributes

- Elements with attributes:

  $<$`aTagname attr`$_1$ `= "val`$_1$`"...attr`$_k$ `= "val`$_k$`"`$>$, $k \geq 1$.

- *empty* element with attributes:

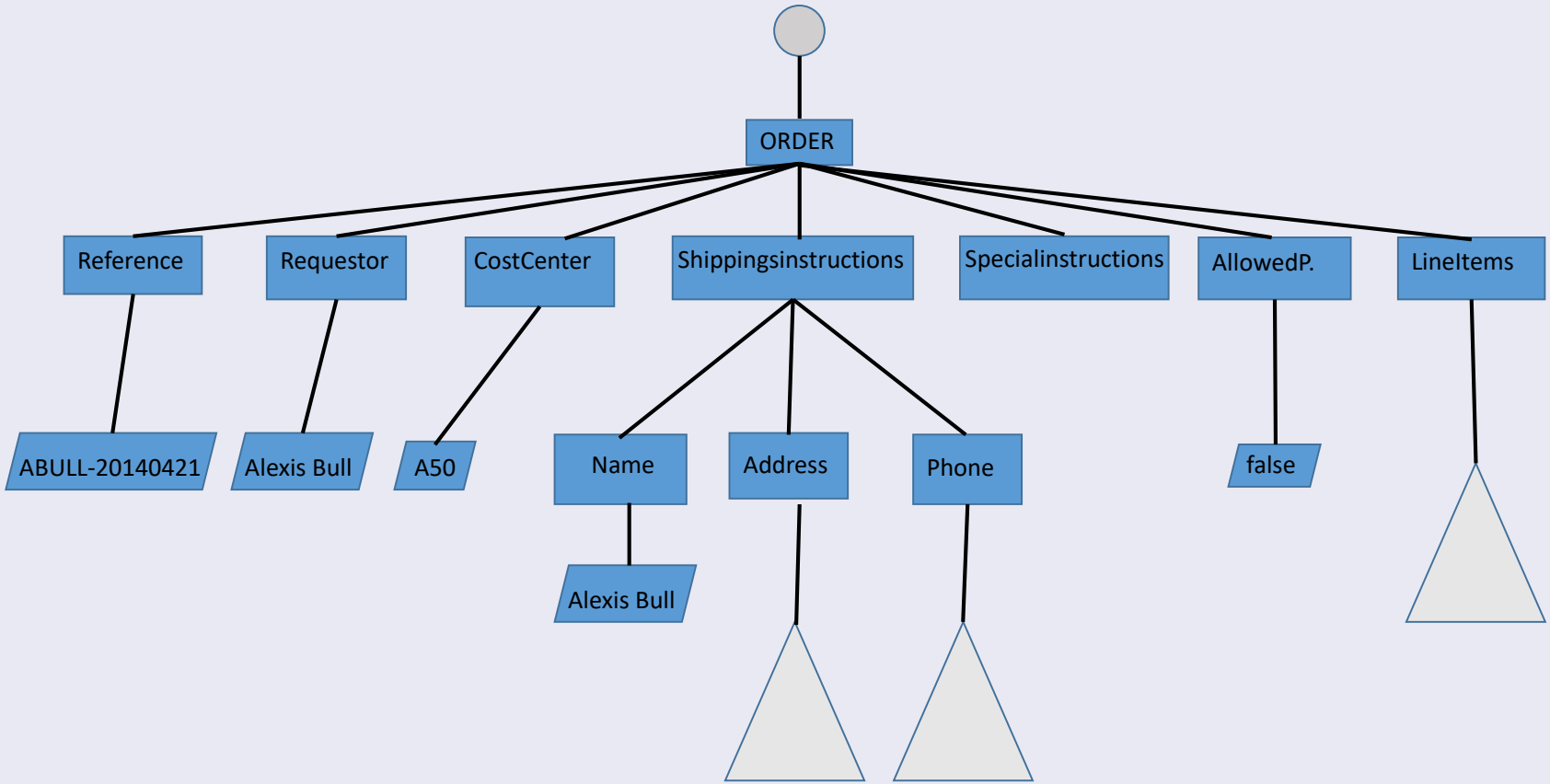
  $<$`aTagname attr`$_1$ `= "val`$_1$`"...attr`$_k$ `= "val`$_k$`"/`$>$, $k \geq 1$.

- Each element may contain to each attribute at most one value.

Example:

```
<Order PONumber="1600/>.
```

XML representation structure: XML-tree



Document-order

- Tags of a document are ordered; the *document-order* is defined by the sequence in which the start-tags inside the document appear.
- Attributes are not ordered; there does not exist a document-order on attributes.
- A XML-tree is *ordered*, if depth-first search of the tree reproduces the document-order of the respective document.
- To each ordered XML-tree there exists a textual representation called *serialization*, which reflects the document-order.

  Note: *A node in the tree represents the corresponding elements start- and end-tag; the element content and the tags and contents of the respective subtree are included.*

Name Spaces

XML-documents may contain other XML-documents as element-content. For example, a message written in XML may contain an XML-document as its data. Name conflicts may appear!

- Element- and attribute-names get a prefix denoting the intended (*name space*).
- Name spaces are identified by URI's.

  A namespace URI in general cannot be used to look up further describing information - using an URI is to simply give the namespace a unique name. However this does not exclude that the dereferenced web page explains the vocabulary used.

Example: Definition of dbis-namespace

- <... xmlns:dbis="http://www.informatik.uni-freiburg/dbis">
- <dbis:course><dbis:name>ADBIS </dbis:name></dbis:course>
- Default-namespace definition:

  <... xmlns="http://www.informatik.uni-freiburg/dbis">

  Elements and attributes used without prefix are assumed to be taken from the default-namespace.
- Namespace-declarations are valid for the respective element - they may be redefined.

  For default namespaces: *The scope of a default namespace declaration extends from the beginning of the start-tag in which it appears to the end of the corresponding end-tag, excluding the scope of any inner default namespace declarations.*

XML documents can be typed using DTD's (Document Type Definition) or XML Schema (not discussed in this course!)

```
<!DOCTYPE Orders[
<!ELEMENT Orders (Order*)>
<!ELEMENT Order (Reference, ..., ShippingInstructions, ..., LineItems)>
<!ELEMENT Reference (#PCDATA)>
<!ELEMENT ShippingInstructions (Name, Address, Phones?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (Street, ...)>
<!ELEMENT Phones (Phone+)>
<!ELEMENT Phone (PhoneType, PhoneNumber>
.
.
.
<!ELEMENT LineItems (Item+)>
<!ELEMENT Item (ItemNumber, Part)>
<!ELEMENT ItemNumber (#PCDATA)>
<!ELEMENT Part (Description, UnitPrice, UPCCode, Quantity)>
.
.
.
<!ATTLIST Order PONumber ID #REQUIRED> ]>
```

DTD

- Using a DTD, *Element types* and *attribut types* are defined.
- A document which conforms to XML syntax rules is called *well-formed*; if it additionally conforms to its DTD it is called *valid*.
- Definition of element types:

  <!ELEMENT EName Content>,
- Definition of attribute types:

  <!ATTLIST EName $\texttt{Attr}_1$ $\texttt{AttrType}_1$ $\texttt{Att\_val}_1$ ... $\texttt{Attr}_k$ $\texttt{AttrType}_k$ $\texttt{Att\_val}_k$>.
- Element types are global with respect to the corresponding DTD - attribute type definitions are local to the corresponding element type definition.

Element content

- `EMPTY`, `ANY`, `(#PCDATA)`;
- $'('$, $')'$ embraced regular expression over element types using $'|'$, $','$, $'*'$, $'+'$ and $'?'$ in the usual way.

Attribute definition[1] `<!ATTLIST EName Attr1 AttrType1 Att_val1 ...>`.

- `AttrType`:
  - `CDATA`,
  - `("val1"| ...| "valn")`
  - `ID`, `IDREF`, `IDREFS`.
    meaning *ID of the respective element (no two elements of the same document can have the same ID value), ID of another element, a list of the latter.*
- `Att_val` is of one of the following:
  - `#REQUIRED`, `#IMPLIED`, `"val"`, `#FIXED "val"`;
    meaning *required, optional, optional with default value, fixed.*

[1] For examples see `https://www.w3schools.com/xml/xml_dtd_attributes.asp`

1.2. XPath

Towards querying XML documents.

- XPath is a language to locate subtrees inside an XML-tree by means of *location paths*.
- A location path is a sequence of '/'-separated *location steps*.
- Each location step is formed out of an *axis*, a *nodetest* and none or several *predicates*:

axis::node-test[predicate].

Literature:
http://www.w3.org/TR/xpath/ which gives a concise specification of XPath 1.0,
http://www.w3.org/TR/xpath-31/ which specifies the most recent version.

Some simple location paths - also introducing a short notation.

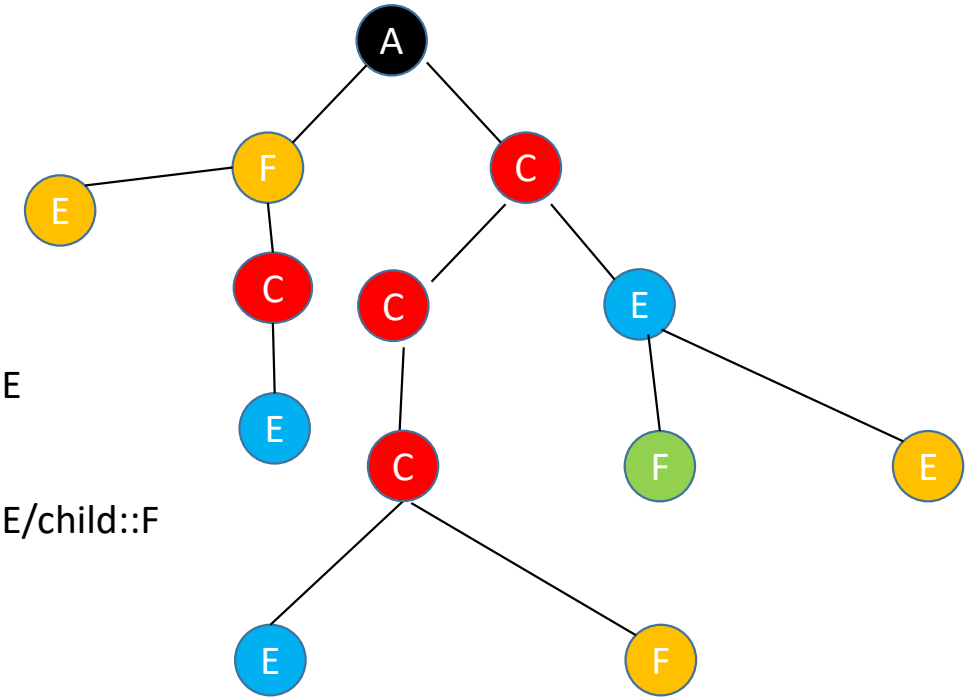
/child::A
/A

/child::A/descendant::C
/A//C

/child::A/descendant::C/child::E
/A//C/E

/child::A/descendant::C/child::E/child::F
/A//C/E/F

node not located



/child::A/descendant::C/child::E/parent::* /A//C/E/..
/child::A/descendant::C/child::E/self::*/parent::* /A//C/E/./..

- A location path P is evaluated relative to a respective *context* and locates a set of nodes, i.e. a set of root nodes of subtrees.
- A context is given by a *context node*, a *context position* and a *context size*.
- A location path starting with '/' is *absolut* and otherwise *relative*.

The context node of an absolut location path is the root node of the document. The root node of the document links to the root element of the document, which is the root node of the respective XML tree.

Absolute location path $P = /p_1/p_2/\ldots/p_n$

- Let n, $n \geq 1$ and p_i, $1 \leq i \leq n$, the i-th location step.
- Let $L_0 = \{r\}$, where r the root node of the document;
  Let $L_i(k)$, $i \geq 1$ be the set of nodes determined by p_i with respect to context node $k \in L_{i-1}$.

  Then, for $i \geq 1$ we define $L_i = \bigcup_{k \in L_{i-1}} L_i(k)$; these are the potential context nodes for the remaining steps.

L_n is called the result of the location path P.

location step: *axis::node-test[predicate]*

Axis:

- *child* axis: the children of the context node,
- *descendant* axis: the descendants of the context node; a descendant is a child or a child of a child and so on,
- *parent* axis: the parent of the context node, if there is one,
- *ancestor* axis: the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on,
- *following-sibling* axis: the right-hand siblings of the context node,
- *preceding-sibling* axis: the left-hand siblings of the context node,
- *following* axis: all nodes that are after the context node in document order, excluding any descendants,
- *preceding* axis: all nodes that are before the context node in document order, excluding any ancestors,
- *attribute* axis: all attributes of the context node,
- *self* axis: the context node itself,
- *descendant-or-self* axis: the context node and its descendants,
- *ancestor-or-self* axis: the context node and its ancestors.

Locating all nodes inside the document.

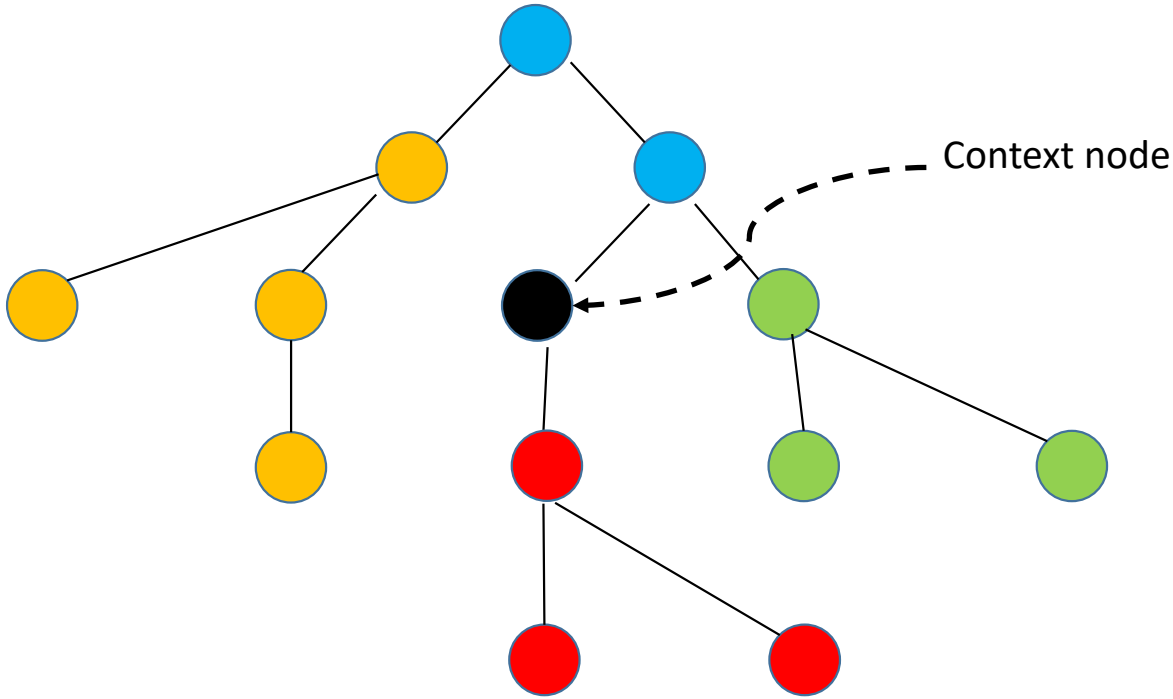
self

decendent

ancestor

following

preceding



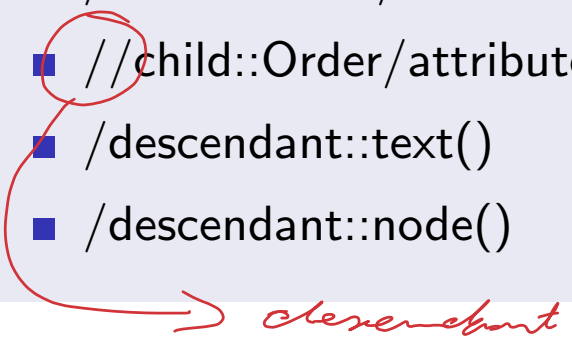
location step: *axis::node-test[predicate]*

Node test

- Element name, resp. attribute name
- '$*$': any attribute or element
- `text()` only text nodes,
- `node()` no restriction

Examples

- /child::*
- /child::*/descendant-or-self::*
- /child::*/descendant::*
- /child::Orders/child::Order/child::Requestor
- //child::Order/attribute::*
- /descendant::text()
- /descendant::node()



location step: *axis::node-test[predicate].*

Predicates

- XPath-expressions which can be assigned a boolean value.

  A location path has value `true()`, if it locates at least one node; otherwise `false()`.
- To evaluate comparison predicates, each node in the result of a location path is replaced by the string built out of the concatenation of all text nodes of the subtree it locates.
- Boolean expressions are built by negation *not*, conjunction *and* or disjunction |.

Examples

- /descendant::*[child::Address]
- /descendant::*[Address]
- /descendant::*[self::Address]
- /child::Orders/child::Order[child::CostCenter= "A50"]/child::LineItems

Comparison-operators: XPath 1.0 Standard

... If both objects to be compared are node-sets, then the comparison will be true if and only if there is a node in the first node-set and a node in the second node-set such that the result of performing the comparison on the string-values of the two nodes is true.

If one object to be compared is a node-set and the other is a number, then the comparison will be true if and only if there is a node in the node-set such that the result of performing the comparison on the number to be compared and on the result of converting the string-value of that node to a number using the number function is true.

If one object to be compared is a node-set and the other is a string, then the comparison will be true if and only if there is a node in the node-set such that the result of performing the comparison on the string-value of the node and the other string is true.

If one object to be compared is a node-set and the other is a boolean, then the comparison will be true if and only if the result of performing the comparison on the boolean and on the result of converting the node-set to a boolean using the boolean function is true. ...

context position

- *Context size* and *context position* are defined relative to a node set.
- The context size is the cardinality of the node set.
- The context position of a node is given by its position relative to document order; for backward axis the reverse document order is the basis.

Examples:

- `//Item[position() = 2]`
- `//Orders/Order[1]//Item[last()]`

Function Library

- Node Set Functions, e.g. `last()`, `position()`, ...,
- String Functions, e.g. `contains(string, string)`, `concat(string, string, string)`, ...,
- Boolean Functions, e.g. `not(boolean)`, `true()`, `false()`, ...,
- Number Functions, e.g. `sum(node-set)`, `round(number)`, ...

Examples:

- `sum(//Item//Quantity)`
- `//*[count(Item) = 2]/parent::*`

XPath is a powerful language. Be aware of the subtleties, in particular when using the short-notation!