INFO-H-509 XML AND WEB TECHNOLOGIES

Lecture 2: XML and XPath

Stijn Vansummeren

February 20, 2019

LECTURE OUTLINE

Unicode

XML and Namespaces

XPath

UNICODE

Unicode (1)

- HTML (and XML) files are represented as text files
- A text file is logically a sequence of characters
- But physically a sequence of bytes
- Several mappings from bytes to characters exist:
  - ASCII
  - EBCDIC
  - Unicode

Unicode aims to cover all characters in all past or present written languages. In its current version, Unicode consists of a repertoire of more than 107,000 characters covering 90 scripts

Unicode (2): characters

Definition

A character is a symbol that appears in a text

Examples

- letters of the alphabet
- pictograms like ©
- accents

Unicode characters are abstract entities:

- LATIN CAPITAL LETTER A
- LATIN CAPITAL LETTER A WITH RING ABOVE
- HIRAGANA LETTER SA
- RUNIC LETTER THURISAZ THURS THORN

Unicode (3): glyphs

Definition

A glyph is a graphical presentation of a character

For instance, the glyph Å represents several characters:

- LATIN CAPITAL LETTER A WITH RING ABOVE
- ANGSTROM SIGN

It can also be described by a sequence of characters:

- LATIN CAPITAL LETTER A COMBINING RING ABOVE

Some characters even result in several glyphs

Unicode (4): code points

Definition

Every Unicode character is assigned a unique number between 0 and 1 114 112, called its code point

For instance:

- the character `HIRAGANA LETTER SA` is assigned the code point 12 373
- Code points 0 through 127 coincide with ASCII

Only 107 000 code points in use today

Unicode (5): character encodings

Definition

A character encoding defines how to interpret a sequence of bytes as a sequence of code points

In Unicode:

- bytes are first parsed into fixed-length code units
- one or more code units may be required to denote a code point

Example character encodings are UTF-8, UTF-16, UTF-32

Unicode (6): UTF-8

- Each code unit is a single byte
- Code points is from 1 to 4 code units
- Code Units between 0 and 127 directly represent the corresponding code points (same as ASCII)

Consider the following sequence of 3 bytes

`0100 0001  0100 0010  0100 0011`

UNICODE (6): UTF-8

- Each code unit is a single byte
- Code points is from 1 to 4 code units
- Code Units between 0 and 127 directly represent the corresponding code points (same as ASCII)

Consider the following sequence of 3 bytes

	0100 0001	0100 0010	0100 0011
decimal	65	66	67

UNICODE (6): UTF-8

- Each code unit is a single byte
- Code points is from 1 to 4 code units
- Code Units between 0 and 127 directly represent the corresponding code points (same as ASCII)

Consider the following sequence of 3 bytes

	`0100 0001`	`0100 0010`	`0100 0011`
decimal	65	66	67
glyphs	A	B	C

UNICODE (6): UTF-8

- For a code unit > 127, the bit masks 110XXXXX, 1110XXXX, and 11110XXX are used to indicate that the code point comprises 2, 3, respectively 4 units.
- The remaining units must have bit mask 10XXXXXX

Consider the following sequence of 3 bytes

```
1110 0011  1000 0001  1001 0101
```

UNICODE (6): UTF-8

- For a code unit > 127, the bit masks 110XXXXX, 1110XXXX, and 11110XXX are used to indicate that the code point comprises 2, 3, respectively 4 units.
- The remaining units must have bit mask 10XXXXXX

Consider the following sequence of 3 bytes

$$1110\,0011 \quad 1000\,0001 \quad 1001\,0101$$

- The first unit indicates that the code point comprises 3 units

UNICODE (6): UTF-8

- For a code unit > 127, the bit masks 110XXXXX, 1110XXXX, and 11110XXX are used to indicate that the code point comprises 2, 3, respectively 4 units.
- The remaining units must have bit mask 10XXXXXX

Consider the following sequence of 3 bytes

1110 0011 10 00 0001 10 01 0101

- The first unit indicates that the code point comprises 3 units
- We then interpret the non-bitmask bits of the 3 units as one binary number

0011 0000 0101 0101

UNICODE (6): UTF-8

- For a code unit > 127, the bit masks 110XXXXX, 1110XXXX, and 11110XXX are used to indicate that the code point comprises 2, 3, respectively 4 units.
- The remaining units must have bit mask 10XXXXXX

Consider the following sequence of 3 bytes

`1110 0011  1000 0001  1001 0101`

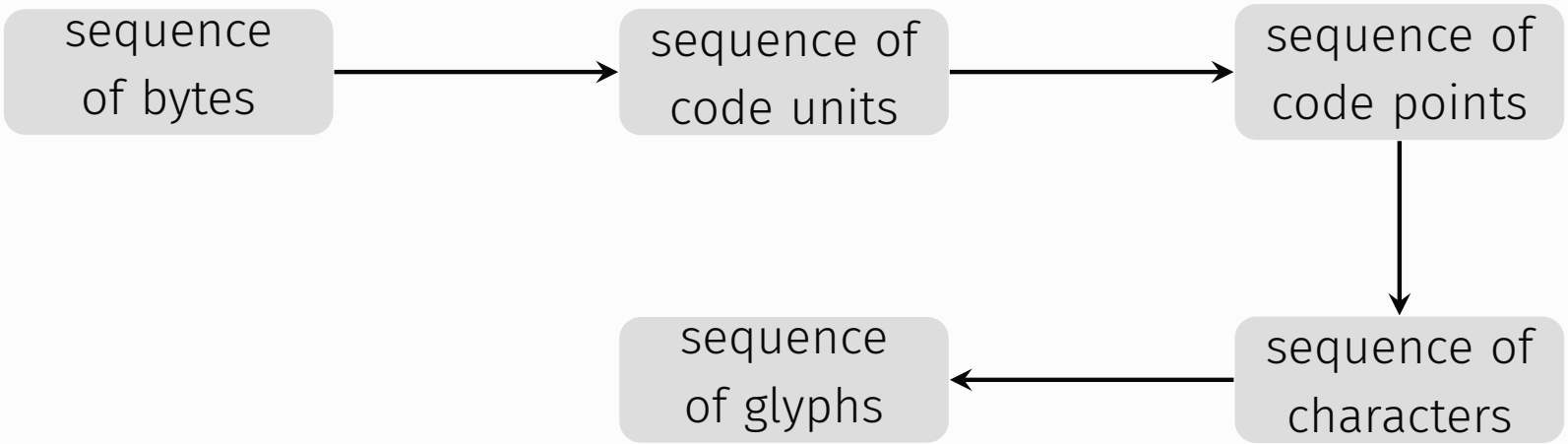
- The first unit indicates that the code point comprises 3 units
- We then interpret the non-bitmask bits of the 3 units as one binary number

`0011 0000 0101 0101`

- This is 12 373 in decimal, the code point for `HIRAGANA LETTER SA`

UNICODE (7)

The big picture:



Other encodings:

- See book for details on UTF-16 and UTF-32
- Another popular character encoding is ISO-8859-1 (aka Latin-1) with only 256 code points. It coincides with ASCII on code points 0-127, but cannot represent general Unicode

UNICODE (8): USE IN HTML

```
<meta http-equiv="Content-Type" content="text/html;
                  charset=ISO-8859-1">
```

XML AND NAMESPACES

WHAT IS XML?

XML: the eXtensible Markup Language

- In essence a data exchange format
- Each application domain is free to chose its own markup tags
- It can also be viewed as a framework for defining markup languages
- There is a common set of generic tools for processing XML documents
- Developed by World Wide Web Consortium (W3C), standardized in 1998.

THE TEXTUAL REPRESENTATION OF XML: AN EXAMPLE

By means of

Online demonstration

THE TEXTUAL REPRESENTATION OF XML: SOME KEY POINTS

- Always starts with the XML Declaration

```
<?xml version="1.1" encoding="UTF-8"?>
```

THE TEXTUAL REPRESENTATION OF XML: SOME KEY POINTS

- Always starts with the XML Declaration

```
<?xml version="1.1" encoding="UTF-8"?>
```

- Consists mainly of Element start and end tags

```
<bla ...> ... </bla>
```

  - Start and end tags must match and nest properly!

    OK: `<x> <y> </y> </x>`

    Not OK: `</z> <x> <y> </x> </y>`

  - Short hand for empty elements: `<bla/>`

THE TEXTUAL REPRESENTATION OF XML: SOME KEY POINTS

- Always starts with the XML Declaration

```
<?xml version="1.1" encoding="UTF-8"?>
```

- Consists mainly of Element start and end tags

```
<bla ...> ... </bla>
```

  - Start and end tags must match and nest properly!

    OK: `<x> <y> </y> </x>`

    Not OK: `</z> <x> <y> </x> </y>`

  - Short hand for empty elements: `<bla/>`

- Attributes: `name="value"` in start tags

THE TEXTUAL REPRESENTATION OF XML: SOME KEY POINTS

- Always starts with the XML Declaration

```
<?xml version="1.1" encoding="UTF-8"?>
```

- Consists mainly of Element start and end tags

```
<bla ...> ... </bla>
```

  - Start and end tags must match and nest properly!

    OK: `<x> <y> </y> </x>`

    Not OK: `</z> <x> <y> </x> </y>`

  - Short hand for empty elements: `<bla/>`

- Attributes: `name="value"` in start tags

- Always exactly one root element

THE TEXTUAL REPRESENTATION OF XML: SOME KEY POINTS

- Always starts with the XML Declaration

```
<?xml version="1.1" encoding="UTF-8"?>
```

- Consists mainly of Element start and end tags

```
<bla ...> ... </bla>
```

  - Start and end tags must match and nest properly!

    OK: `<x> <y> </y> </x>`

    Not OK: `</z> <x> <y> </x> </y>`

  - Short hand for empty elements: `<bla/>`

- Attributes: `name="value"` in start tags

- Always exactly one root element

Well-formedness can be checked by tools

THE TEXTUAL REPRESENTATION OF XML: SPECIAL CONSTRUCTS

- Comment tags: allow you to write some meta-information in the XML file. This is normally ignored by all XML processing applications.

```
<!– bla –>
```

- Processing instructions: allow you to specify specific instructions to specific processors. Each has a target and a value.

```
<?target value?>
```

THE TEXTUAL REPRESENTATION OF XML: MORE CONSTRUCTS

Special characters can be referenced by character references

- These take the form `&#XXXX;`

```
<bla> Copyright &#169; 2005 </bla>
```

- `<` and `>` and `&` denote < and > and & respectively

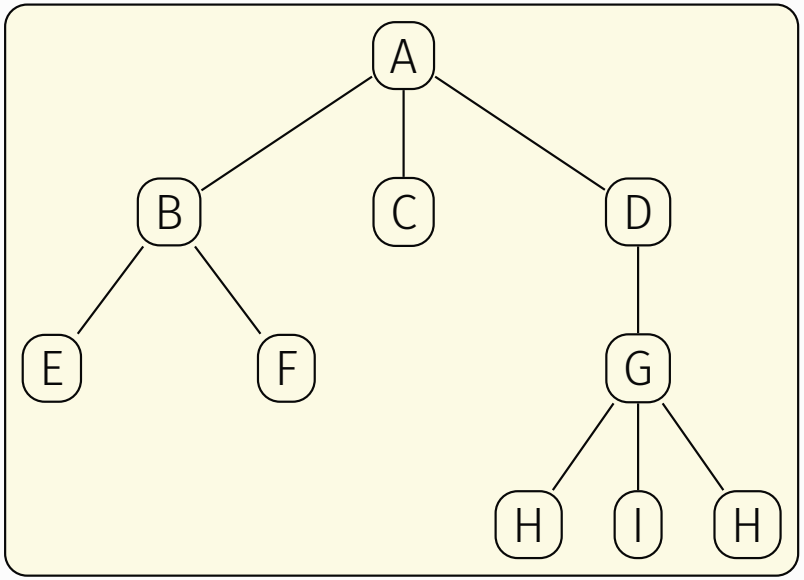
CDATA sections allow text that doesn't need to be escaped by `&lt` and `&gt`

```
<![CDATA[ <this is not a tag> ]]>
```

REFRESH YOUR MEMORY ABOUT TREES

Do you know the following terms?

- node, edge
- root, leaf
- child, parent
- sibling (ordered trees), ancestor, descendant

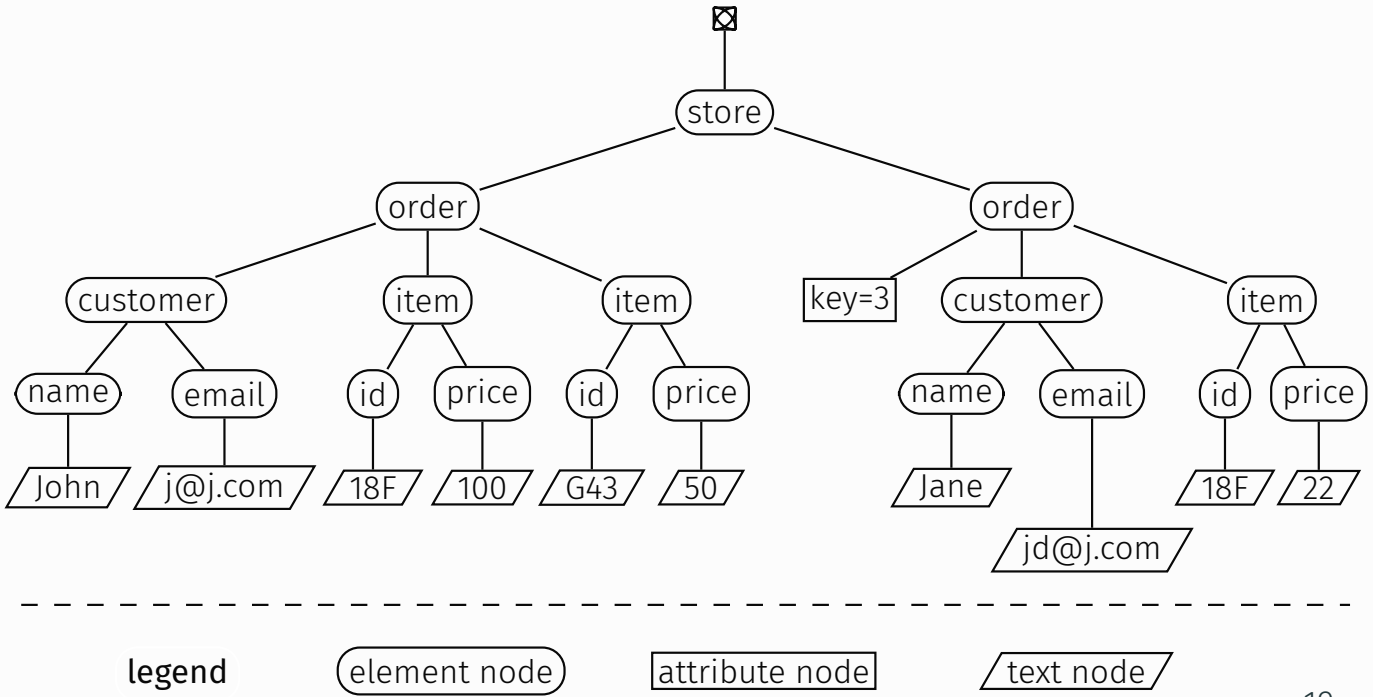


THE XML CONCEPTUAL DATA MODEL: TREES

```
<?xml version="1.1" encoding="UTF-8"?>
<store>
 <order>
   <customer>
      <name>John</name>
    <email>j@j.com</email>
   </customer>
   <item>
     <id> 18F </id>
     <price> 100 </price>
   </item>
   <item>
     <id> G43 </id>
     <price> 50 </price>
   </item>
 </order>
 <order key="3">
   <customer>
      <name>Jane</name>
    <email>jd@j.com</email>
   </customer>
   <item>
     <id> 18F </id>
     <price> 22 </price>
   </item>
 </order>
</store>
```

Conceptually, an XML document is an ordered tree with different types of nodes

- document structure = tree structure
- document data = leaf of tree
- also comment nodes, processing instructions, ... (not shown)



Definition

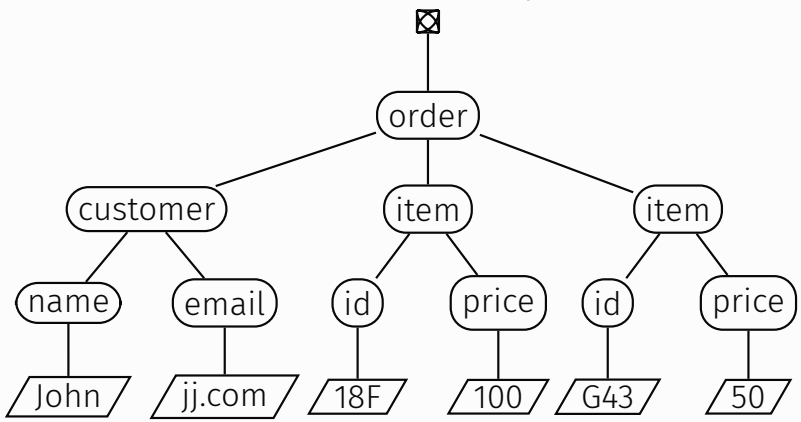
An XML parser is an algorithm that given the textual XML document, constructs its tree representation

On file

XML Parser

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
   <customer>
      <name>John</name>
      <email>j@j.com</email>
   </customer>
   <item>
     <id> 18F </id>
     <price> 100 </price>
   </item>
   <item>
     <id> G43 </id>
     <price> 50 </price>
   </item>
</order>
```

In memory



THE XML CONCEPTUAL DATA MODEL: CAUTION!

There are multiple data models for XML:

- XQuery and XPath data model (used here)
- The post-validation infoset
- DOM, JDOM
- ...

We use the XPath data model unless specified otherwise!

EXAMPLE: A RECIPY MARKUP LANGUAGE

```
<?xml version="1.0" encoding="UTF-8"?>
<collection>
 <description>Recipes suggested by Jane Dow</description>

 <recipe id="r117">
   <title>Rhubarb Cobbler</title>
   <date>Wed, 14 Jun 95</date>

   <ingredient name="diced rhubarb" amount="2.5" unit="cup"/>
   <ingredient name="sugar" amount="2" unit="tablespoon"/>
   <ingredient name="fairly ripe banana" amount="2"/>
   <ingredient name="cinnamon" amount="0.25" unit="teaspoon"/>
   <ingredient name="nutmeg" amount="1" unit="dash"/>

   <preparation>
      <step>Combine all and use as cobbler, pie, or crisp.</step>
   </preparation>

   <comment> Rhubarb Cobbler made it with ... </comment>

   <nutrition calories="170" fat="28%"
              carbohydrates="58%" protein="14%"/>
   <related ref="42">Garden Quiche</related>
  </recipe>
</collection>
```

Rough Classification:

- Data-oriented languages (e.g configuration files, CML)
- Document-oriented languages (e.g. XHTML)
- Protocols and programming languages (e.g SOAP, WSDL, XSLT)
- Hybrids

EXAMPLE: XHTML

XHTML is the XML version of HTML

- XHMTL must be a well-formed XML document (e.g. tags must match and nest properly)
- Ordinary HTML is typically “invalid” and hence renders differently on different browsers.

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Hello world!</title></head>
  <body>
    <h1>This is a heading</h1>
    This is some text.
  </body>
</html>
```

EXAMPLE: CHEMICAL MARKUP LANGUAGE CML

```
<?xml version="1.0" encoding="UTF-8"?>
<molecule id="METHANOL">
 <atomArray>
  <stringArray builtin="id">a1 a2 a3 a4 a5 a6</stringArray>
  <stringArray builtin="elementType">C O H H H H</stringArray>
  <floatArray builtin="x3" units="pm">
    -0.748 0.558 ...
  </floatArray>
  <floatArray builtin="y3" units="pm">
    -0.015 0.420 ...
  </floatArray>
  <floatArray builtin="z3" units="pm">
    0.024 -0.278 ...
  </floatArray>
 </atomArray>
</molecule>
```

EXAMPLE: EBXML

```
<?xml version="1.0" encoding="UTF-8"?>
<MultiPartyCollaboration name="DropShip">
 <BusinessPartnerRole name="Customer">
  <Performs initiatingRole='//binaryCollaboration[@name="Firm Order"]/
                            InitiatingRole[@name="buyer"]' />
 </BusinessPartnerRole>
 <BusinessPartnerRole name="Retailer">
  <Performs respondingRole='//binaryCollaboration[@name="Firm Order"]/
                            RespondingRole[@name="seller"]' />
  <Performs initiatingRole='//binaryCollaboration[...]/
                            InitiatingRole[@name="buyer"]' />
 </BusinessPartnerRole>
 <BusinessPartnerRole name="DropShip Vendor">
  ...
 </BusinessPartnerRole>
</MultiPartyCollaboration>
```

EXAMPLE: THML

```
<?xml version="1.0" encoding="UTF-8"?>
<h3 class="s05" id="One.2.p0.2">Having a Humble Opinion of Self</h3>
<p class="First" id="One.2.p0.3">EVERY man naturally desires knowledge
  <note place="foot" id="One.2.p0.4">
    <p class="Footnote" id="One.2.p0.5"><added id="One.2.p0.6">
      <name id="One.2.p0.7">Aristotle</name>, Metaphysics, i. 1.
    </added></p>
  </note>;
  but what good is knowledge without fear of God? Indeed a humble
  rustic who serves God is better than a proud intellectual who
  neglects his soul to study the course of the stars.
  <added id="One.2.p0.8"><note place="foot" id="One.2.p0.9">
    <p class="Footnote" id="One.2.p0.10">
      Augustine, Confessions V. 4.
    </p>
  </note></added>
</p>
```

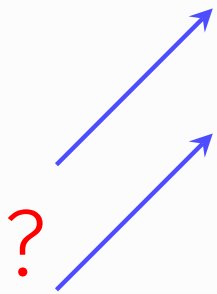
XML NAMESPACES (1): MOTIVATION

Consider an XML language for widgets. The **<info>** element contains information in XHTML markup.

```
<widget type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <head>
      <title>Description of gadget</title>
    </head>
    <body>
      <h1>Gadget</h1>
      A gadget contains a big gizmo
    </body>
  </info>
</widget>
```

XML NAMESPACES (1): MOTIVATION

Consider an XML language for widgets. The **<info>** element contains information in XHTML markup.




```
<widget type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <head>
      <title>Description of gadget</title>
    </head>
    <body>
      <h1>Gadget</h1>
      A gadget contains a big gizmo
    </body>
  </info>
</widget>
```

XML NAMESPACES (1): MOTIVATION

Consider an XML language for widgets. The **<info>** element contains information in XHTML markup.


```
<widget type="gadget">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info>
    <head>
      <title>Description of gadget</title>
    </head>
    <body>
      <h1>Gadget</h1>
      A gadget contains a big gizmo
    </body>
  </info>
</widget>
```

- When combining languages, element names may become ambiguous
- A general solution?

XML NAMESPACES (2): THE IDEA

- Assign a URI to every (sub-)language

  e.g. `http://www.w3.org/1999/xhtml` for XTML 1.0

- Qualify element names with URIs:

  {http://www.w3.org/1999/xhtml}head

XML NAMESPACES (3): ACTUAL SOLUTION

Namespace declarations bind URIs to prefixes

```
<... xmlns:foo="http://www.w3.org/TR/xhtml1">
   ...
   <foo:head>...</foo:head>
   ...
</...>
```

- Lexical scope
- Default namespace (no prefix) declared with xmlns="..."
- Attribute names can also be prefixed

XML NAMESPACES (4): EXAMPLE

Namespace declarations bind URIs to prefixes

```
<widget type="gadget" xmlns="http://www.widget.inc">
  <head size="medium"/>
  <big><subwidget ref="gizmo"/></big>
  <info xmlns:xhtml="http://www.w3.org/TR/xhtml1">
    <xhtml:head>
      <xhtml:title>Description of gadget</xhtml:title>
    </xhtml:head>
    <xhtml:body>
    <xhtml:h1>Gadget</xhtml:h1>
      A gadget contains a big gizmo
    </xhtml:body>
  </info>
</widget>
```

Namespace map: for each element, maps prefixes to URIs

XPATH

WHAT IS XPATH?

XPath is:

- A flexible notation for navigating in trees
- Widely used as a basic component in many XML-related standards:
  - XML Schema: uniqueness and scope
  - XSLT & XQuery: pattern matching, selection and value computation
  - XLink & XPointer: document relationships

XPath has multiple versions:

- version 1.0 (limited to location paths)
- version 2.0 (focus of this lecture)
- version 3.0 (small changes w.r.t 2.0, standardized in 2014)
- version 3.1 (towards support for JSON, standardized in 2017)

Definition

The basic building block of XPath expressions is the location path, which consists of a sequence of location steps separated by /.

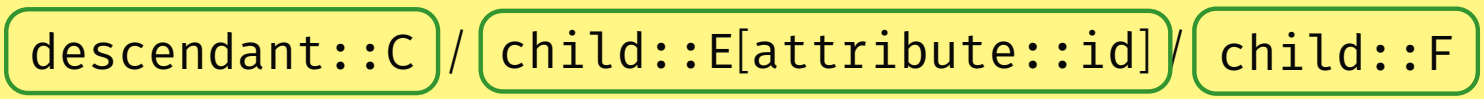
Example location path:

```
descendant::C / child::E[attribute::id]/ child::F
```

Definition

The basic building block of XPath expressions is the location path, which consists of a sequence of location steps separated by /.

Example location path:



location step location step location step

Definition

A location step consists of

- an axis
- a nodetest
- (optionally) some predicates

with the following general syntax

axis :: *nodetest* [*exp1*] [*exp2*] ...

Examples:

- child::rcp:ingredient
- child::rcp:recipe[attribute::id='117']
- attribute::amount

EVALUATING A LOCATION PATH

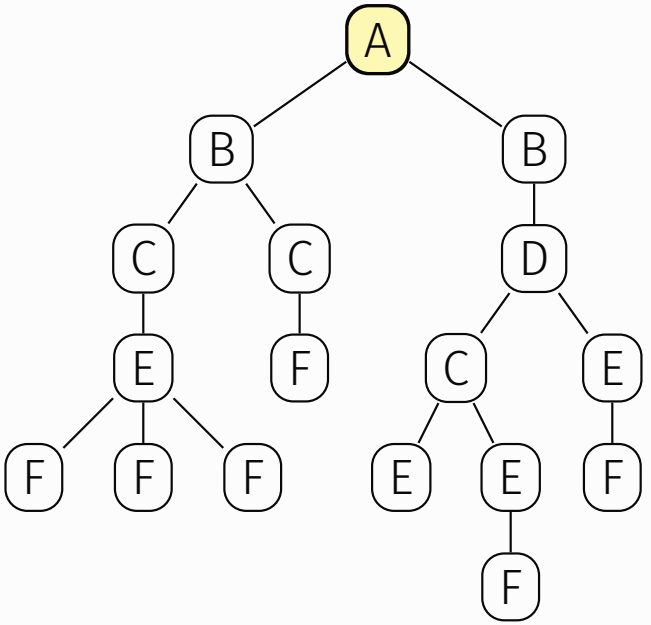
- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

Start with context node

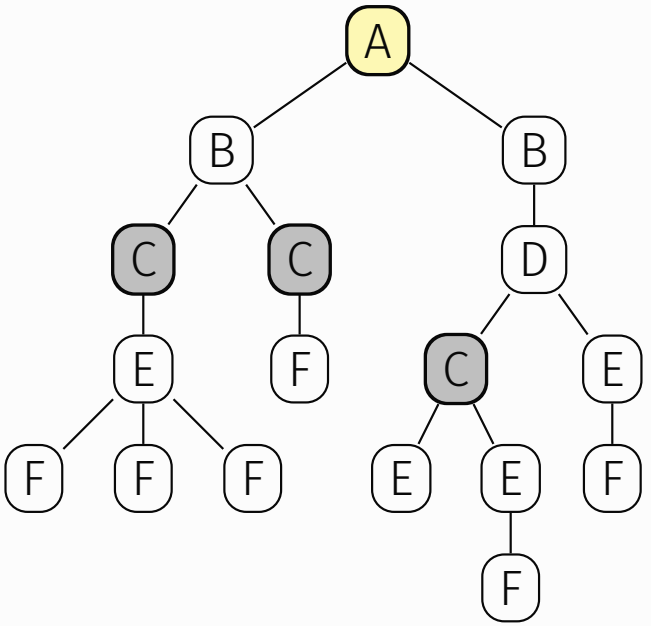


EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

descendant::C

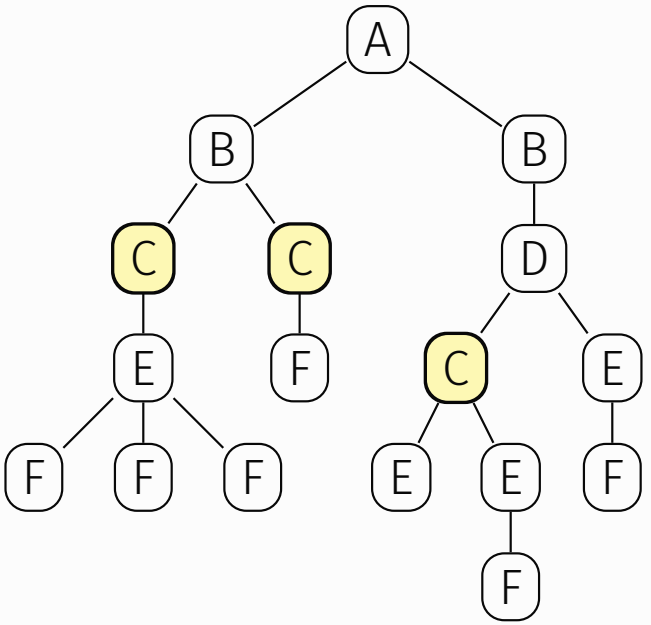


EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

descendant::C/child::E

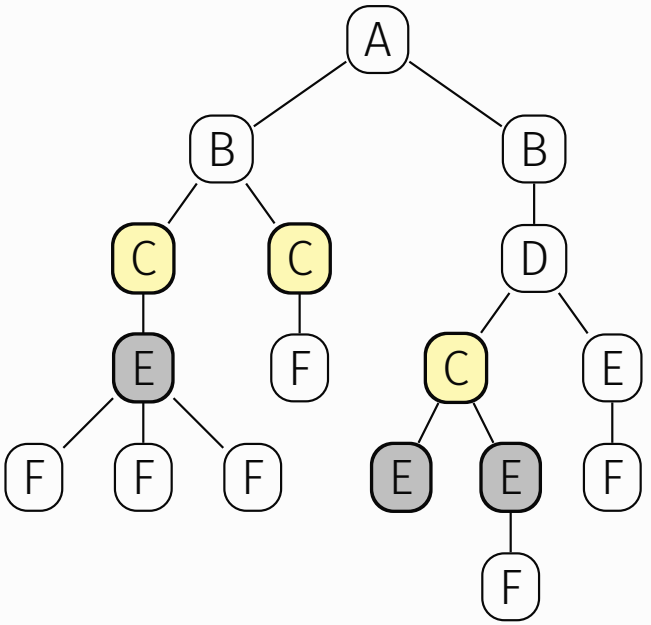


EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

descendant::C/child::E

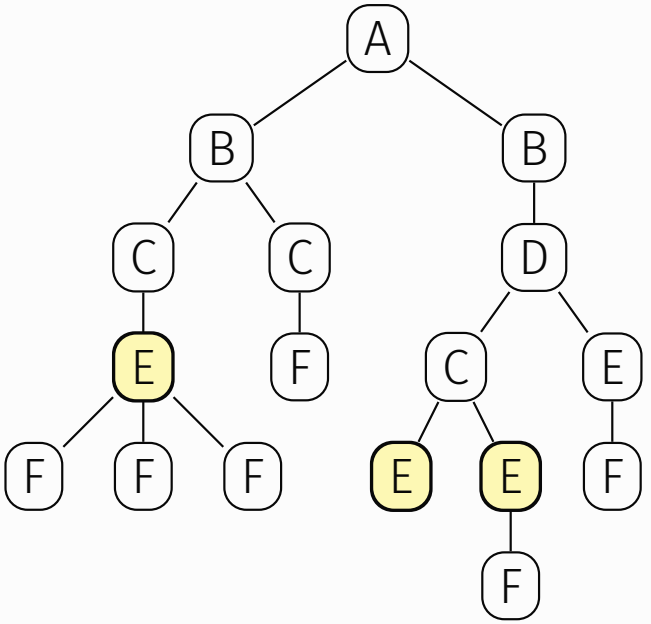


EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

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

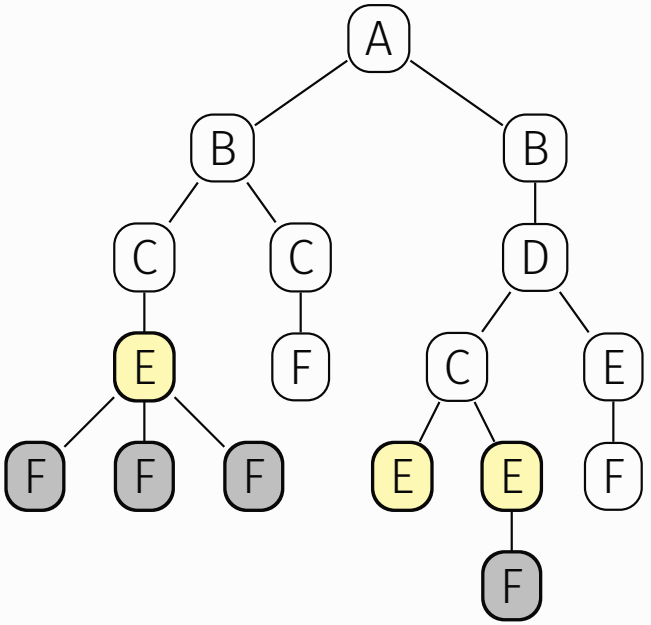


EVALUATING A LOCATION PATH

- Semantically, each step maps a context node into a sequence
- This is easily extended to map sequences of nodes to sequences of nodes
  - each node in the input sequence serves as a context node
  - and is replaced with the result of applying the step
- The path then applies each step in return

Example:

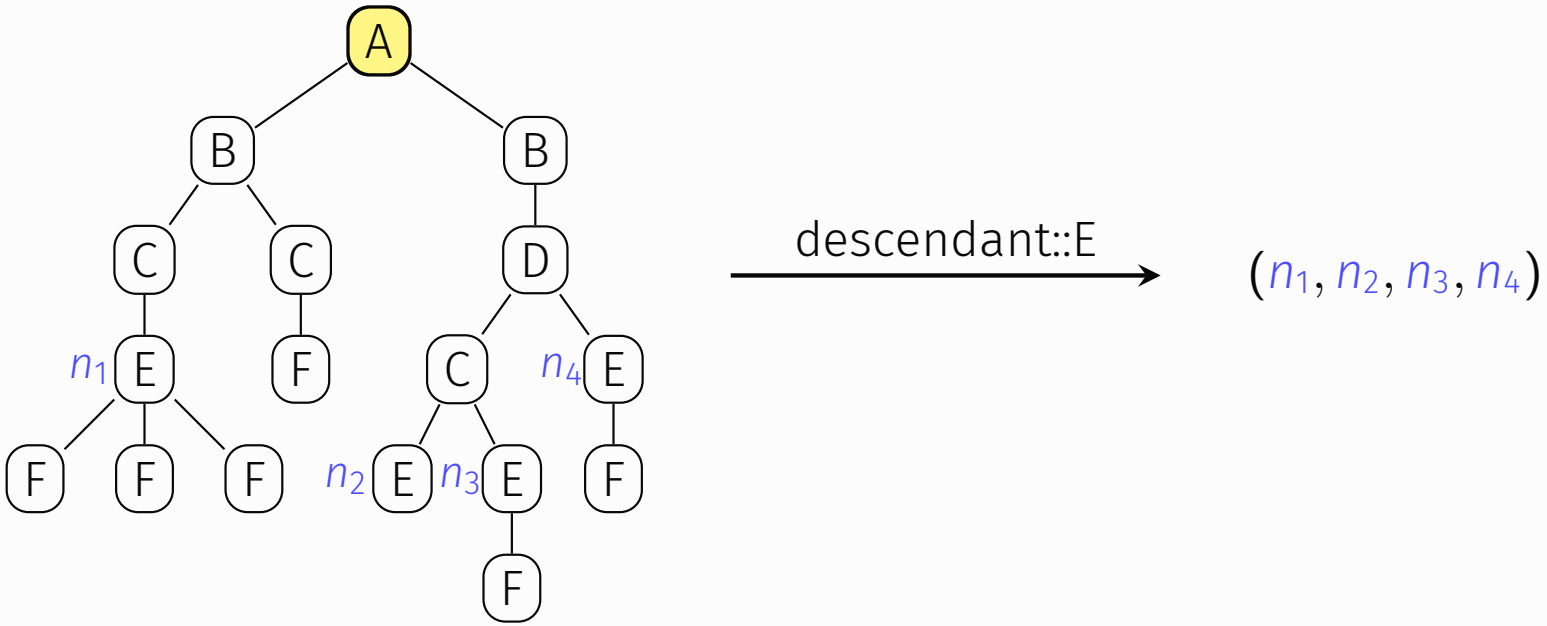
descendant::C/child::E/child::F



EVALUATING A SINGLE LOCATION STEP (1)

Definition

- The input to a location step consists of (1) the tree representation of an XML document and (2) a context
- The output of a location step is a sequence of nodes



EVALUATING A SINGLE LOCATION STEP (2)

Definition

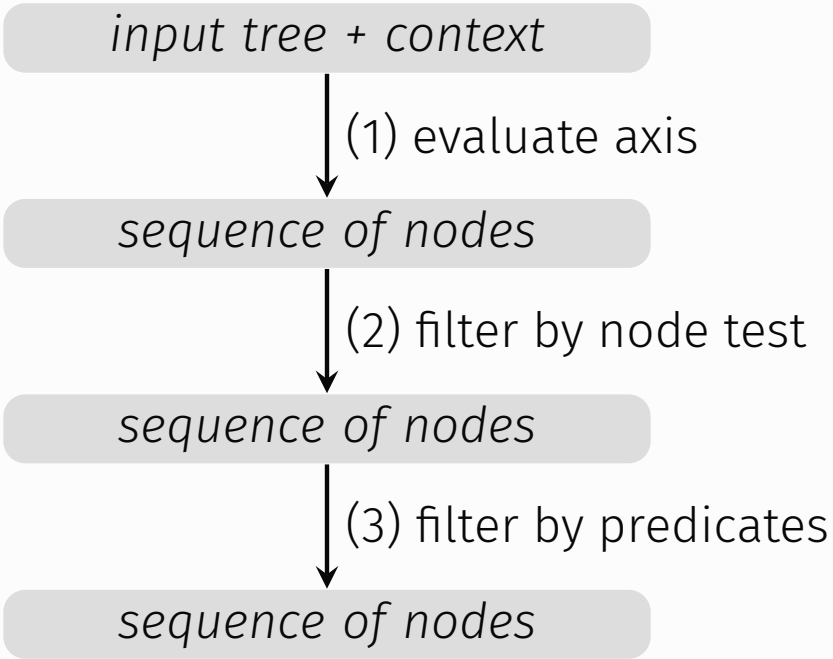
The context of an XPath expression consists of 6 components:

- a context node (a node in the XML tree)
- a context position and size (two natural numbers)
- a set of variable bindings
- a function library (depends on the implementation)
- a set of namespace declarations (mapping prefixes to namespace URIs)

- The application determines the initial context
- If the path starts with '/' then
  - the initial context node is the root
  - the initial position and size are 1

EVALUATING LOCATION STEPS (3)

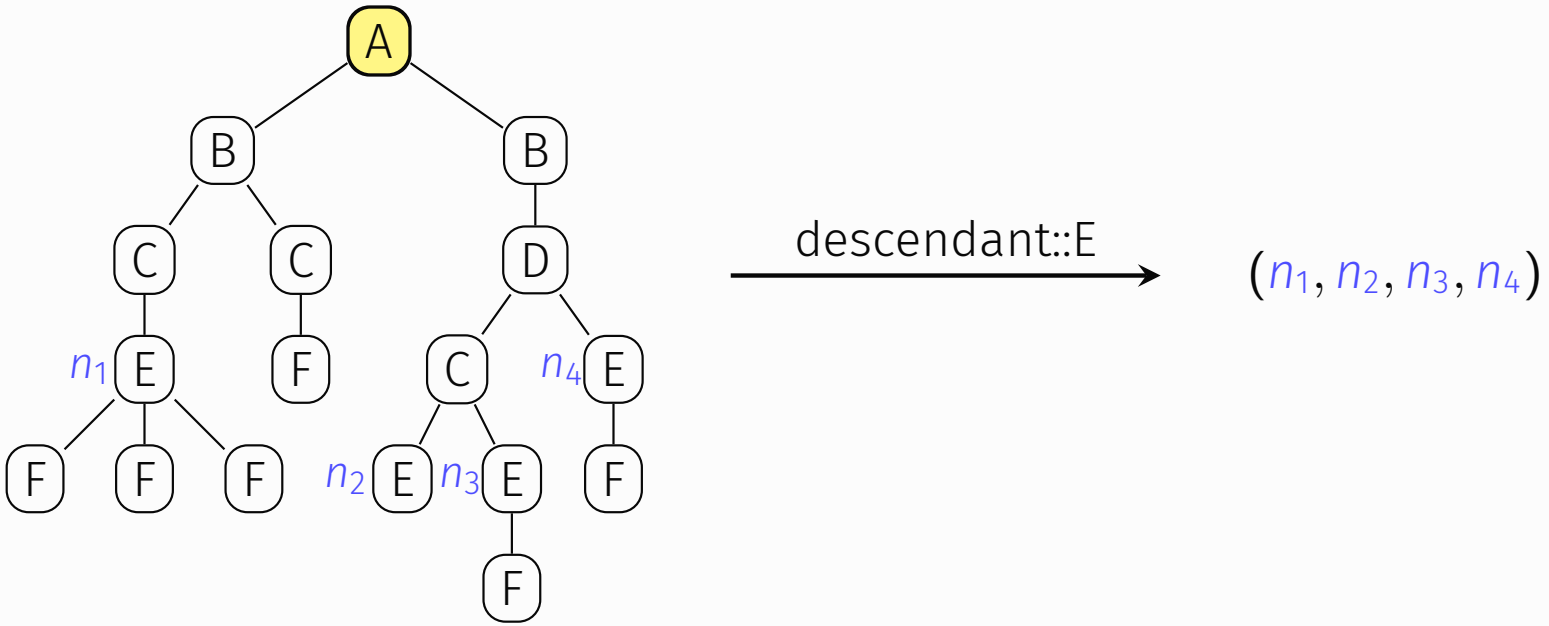
To evaluate *axis* :: *nodetest* [*exp1*] [*exp2*]:



THE ORDER OF NODES IN THE RESULTING SEQUENCE

Definition

- The document order of an XML tree is the order in which node *m* occurs before node *n* if the open tag of *m* occurs before the open tag of *n* in the corresponding text representation.
- The result of a location step and path is always sorted in document order, with duplicate nodes removed.



EVALUATING THE AXIS

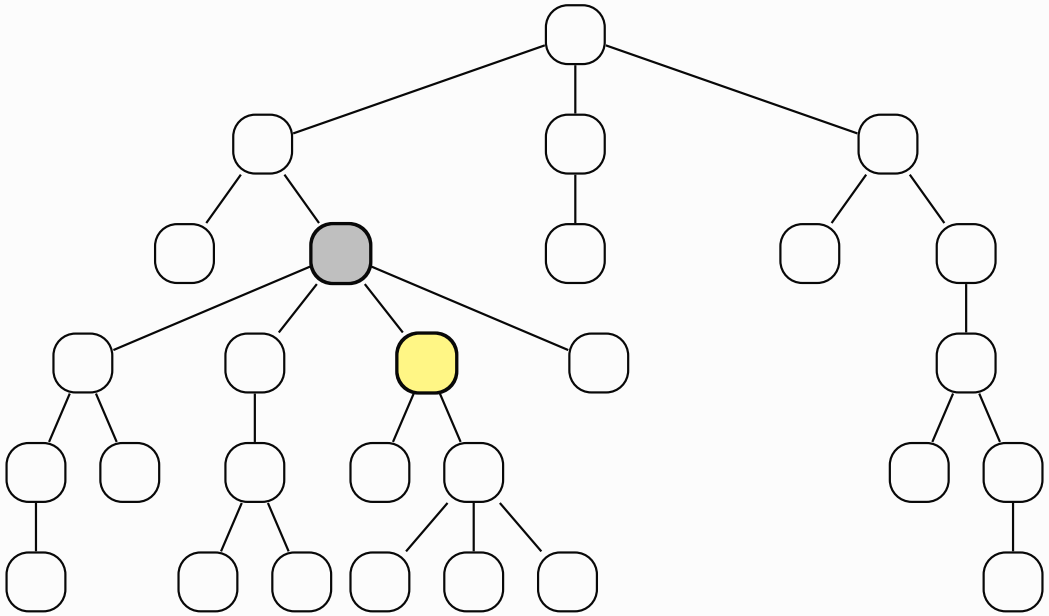
Remember:

- An axis is evaluated relative to the context node
- And evaluates to a sequence of nodes

XPath supports 12 different axes

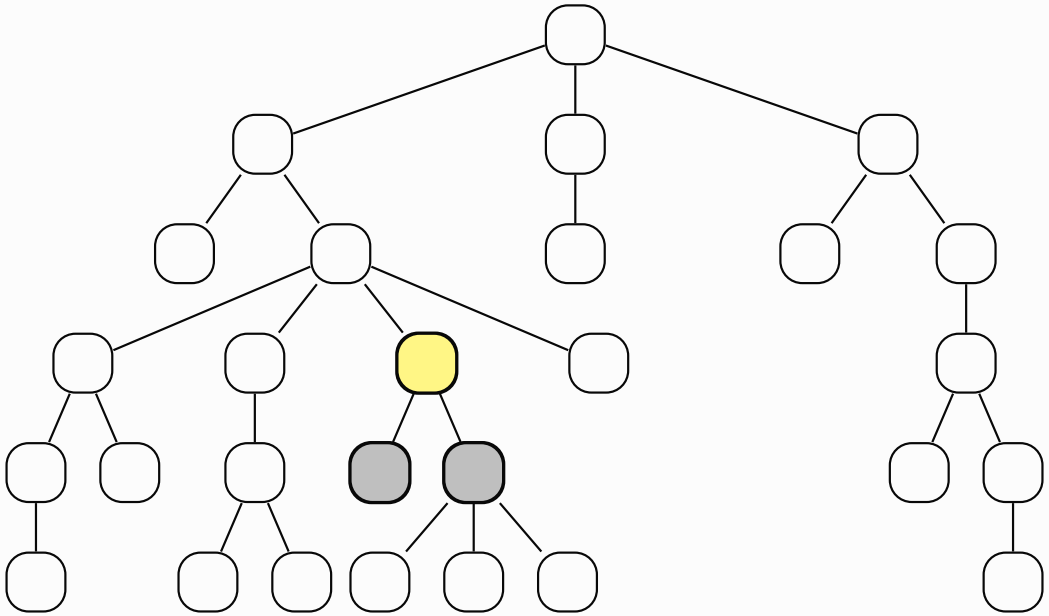
- child
- descendant
- parent
- ancestor
- following-sibling
- preceding-sibling
- attribute
- following
- preceding
- self
- descendant-or-self
- ancestor-or-self

EVALUATING THE AXIS (1): THE PARENT AXIS



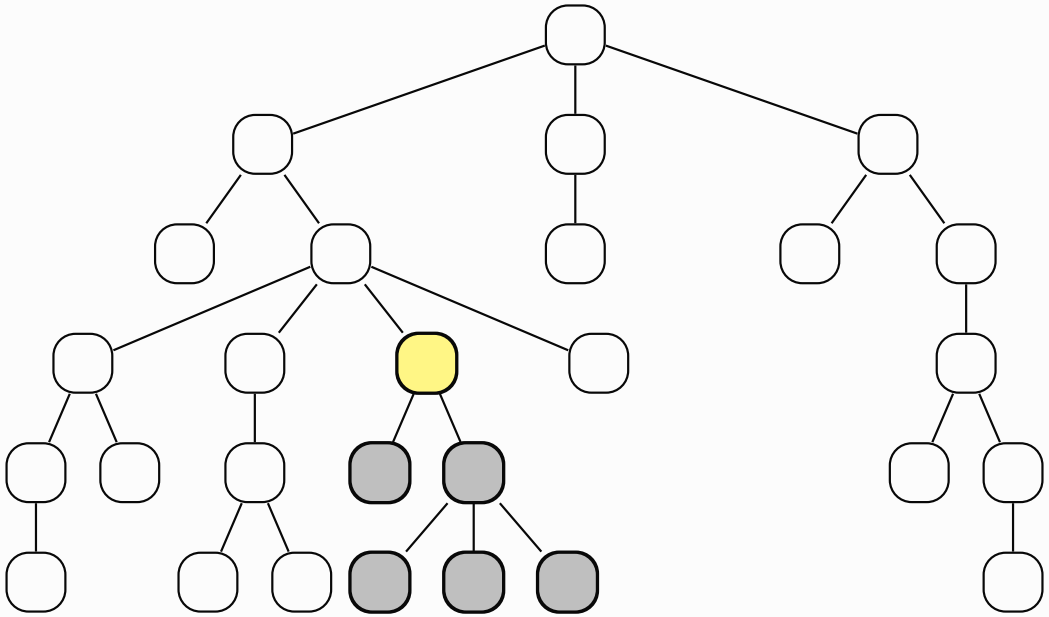
- Selects the unique parent node of the context node.
- Returns the empty sequence if the context node is the root.

EVALUATING THE AXIS (2): THE CHILD AXIS



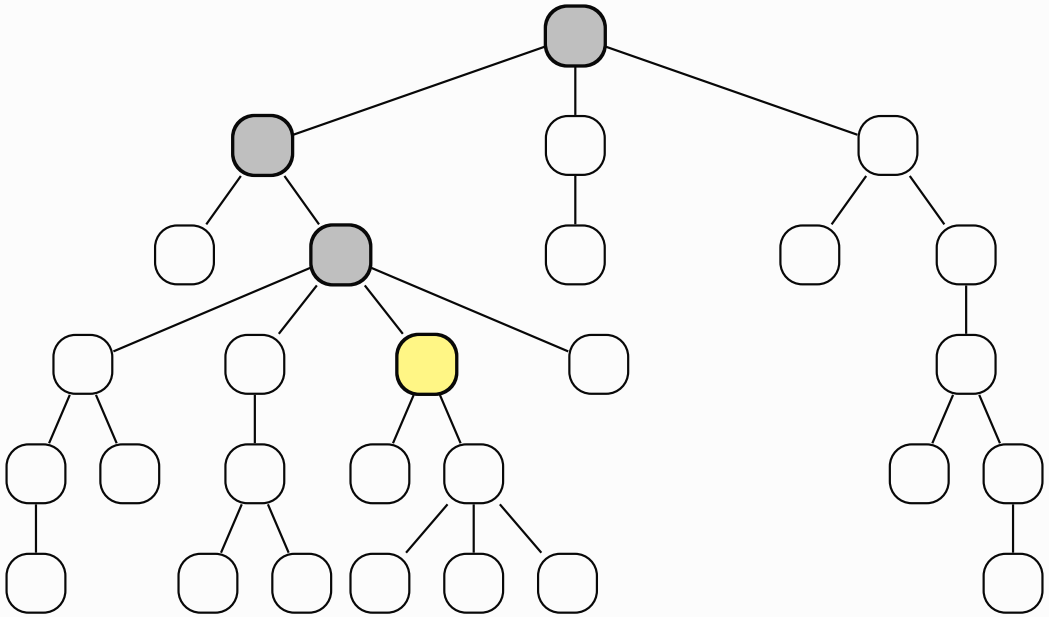
- Selects all children excluding attributes

EVALUATING THE AXIS (3): THE DESCENDANT AXIS



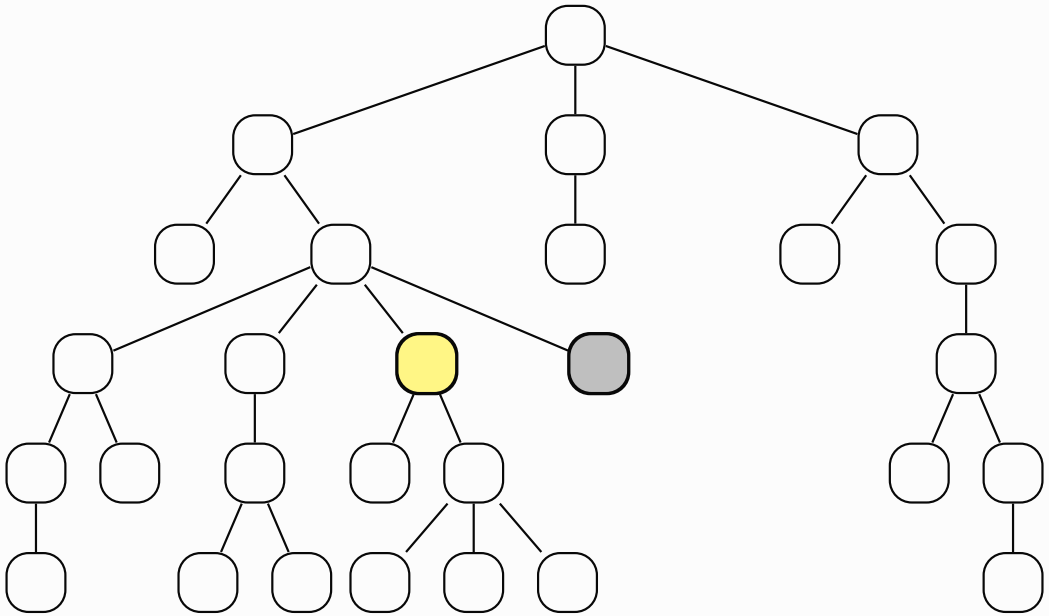
- The transitive closure of the child axis. Selects all descendants excluding attributes.
- The `descendant-or-self` axis returns the concatenation of the `self` and `descendant` sequences.

EVALUATING THE AXIS (4): THE ANCESTOR AXIS



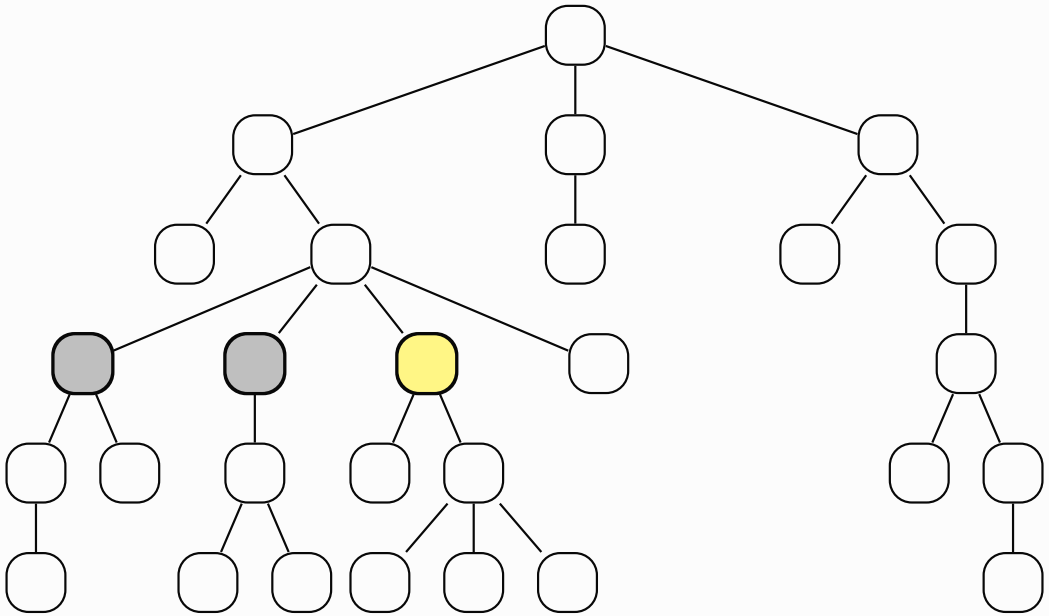
- The transitive closure of the parent axis. All ancestors of the context node, from parent to the root element node.
- The `ancestor-or-self` axis returns the concatenation of the `self` and `ancestor` sequences.

EVALUATING THE AXIS (5): THE FOLLOWING-SIBLING AXIS



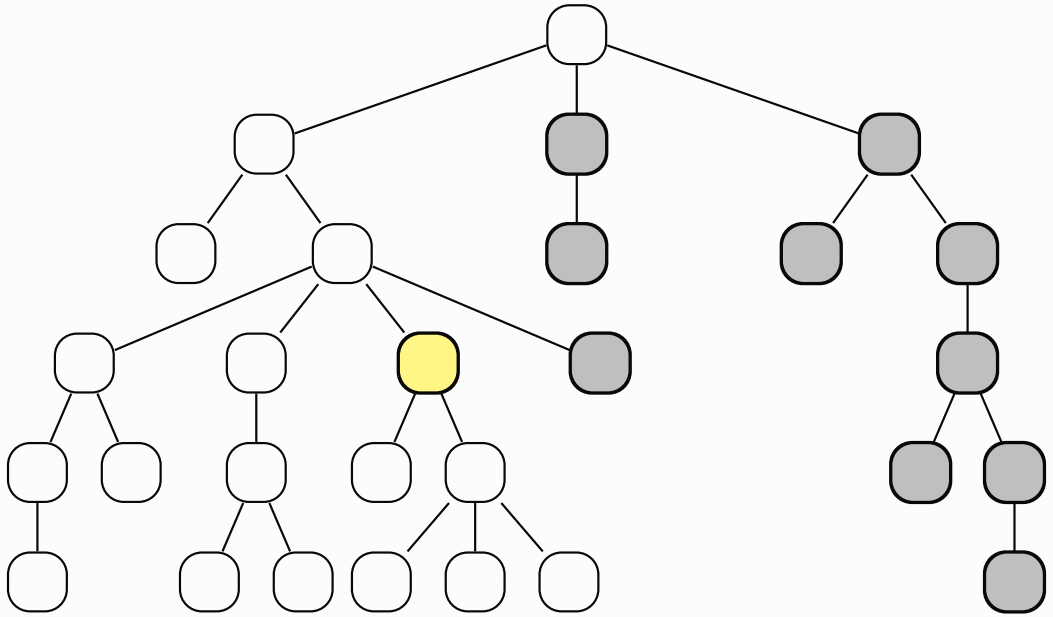
- Returns the right-hand siblings of the context node excluding attributes. It returns the empty sequence if the context node is an attribute.

EVALUATING THE AXIS (6): THE PRECEDING-SIBLING AXIS



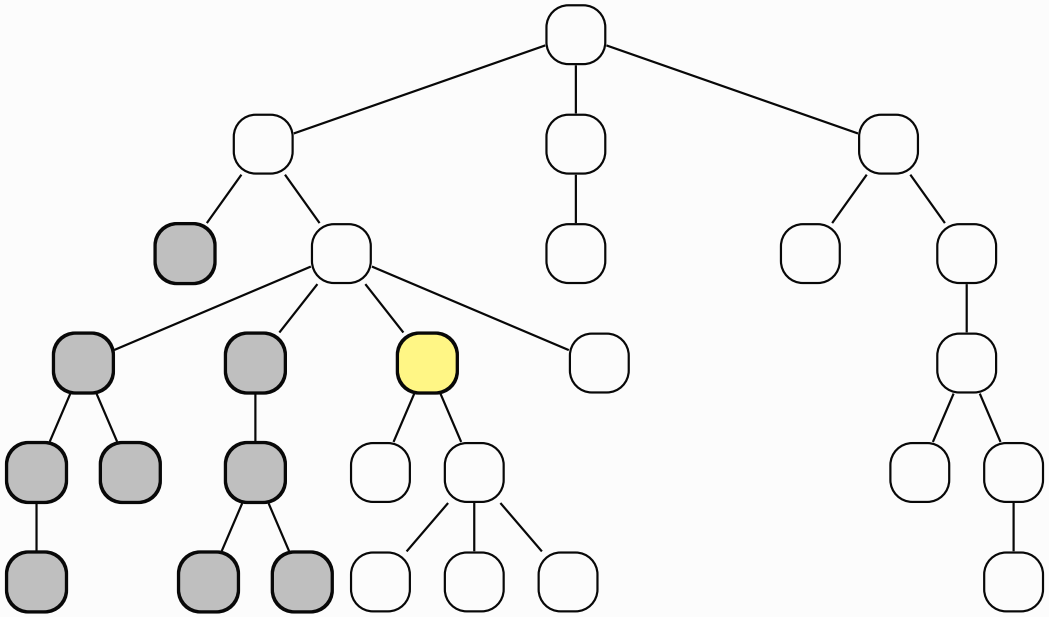
- Returns the left-hand siblings of the context node, excluding attributes. It returns the empty sequence if the context node is an attribute.

EVALUATING THE AXIS (7): THE FOLLOWING AXIS



- Returns all nodes appearing strictly later in document order, but excluding descendants and attribute nodes.

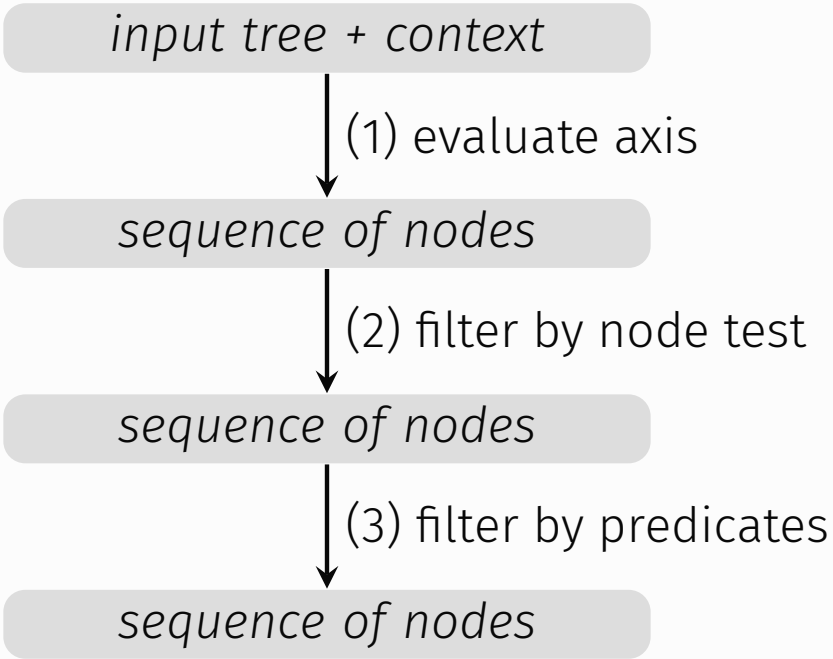
EVALUATING THE AXIS (8): THE PRECEDING AXIS



- Returns all nodes appearing strictly earlier in document order, but excluding ancestors and attribute nodes.

EVALUATING LOCATION STEPS: REMEMBER

To evaluate *axis* :: *nodetest* [*exp1*] [*exp2*]:



NODE TESTS

The node test in a location path takes as input the sequence computed by the location step, and further selects from this sequence all nodes that satisfy the test.

In particular

- `text()` selects all text nodes in the sequence
- `comment()` selects all comment nodes
- `processing-instruction()` selects all processing instruction nodes
- `node()` select all nodes
- `*` selects all element nodes, except when it is used after an `attribute` axis, in which case it selects all attribute nodes
- *name* selects all nodes with the given name (must be a Qualified Name, or QName for short). Text nodes do not have a name!
- **:localname* selects the nodes with the given name in any namespace
- *prefix:** selects nodes as `*`, but only those in the given namespace.

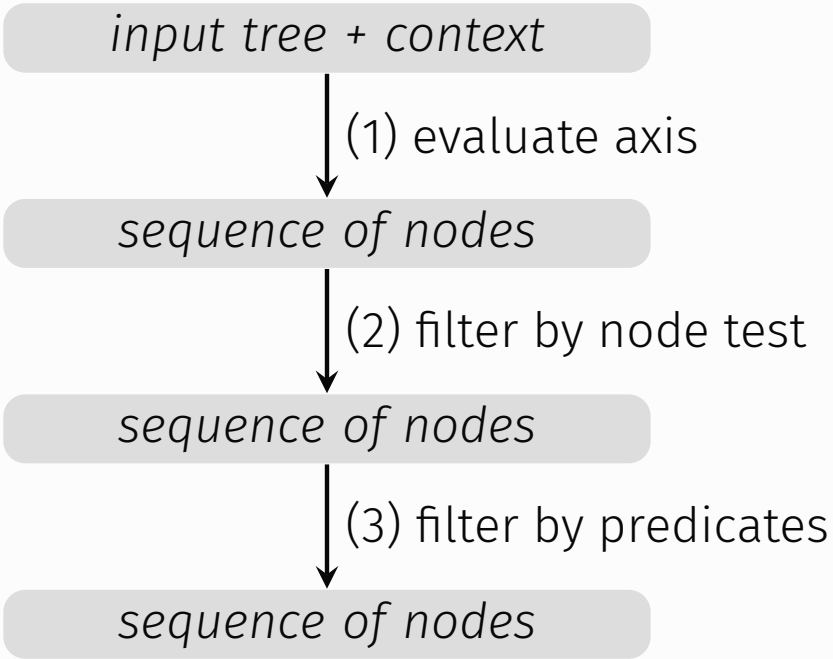
QUIZ

What does location step child::hi select in the following document when evaluated from the root?

```
<bla>
  <hi/> hi
</bla>
```

EVALUATING LOCATION STEPS: REMEMBER

To evaluate *axis* :: *nodetest* [*exp1*] [*exp2*]:



PREDICATES

- The predicates in a location path take as input the sequence computed by the node test. For each node in that sequence, the predicate is evaluated (using the current node as context node). The result is coerced into a boolean:
  - a number yields `true` if it equals the context position
  - a string yields `true` if it is nonempty
  - a sequence yields `true` if it is nonempty

  All nodes that returned `true` are kept, the rest is discarded.
- A predicate can be any XPath expression (see later), it need not be a location step

Examples

- /descendant::recipe[descendant::ingredient]
- /descendant::ingredient[attribute::name='sugar']
- /descendant::recipe[descendant::ingredient[attribute::name='sugar']]
- /descendant::recipe[4]
- /descendant::recipe[position()=4][position()=1]

A NOTE ABOUT AXES DIRECTIONS

Each XPath axis has a direction

- A forward direction means that the input sequence is filtered in document order:
  - child, descendant, following-sibling, following, self, descendant-or-self
- A backward direction means that the input sequence is filtered in reverse document order:
  - parent, ancestor, preceding-sibling, preceding
- Attribute nodes are considered to be unordered. Therefore, the order in which the nodes in the sequence resulting from the `attribute` axis are filtered is implementation-dependent, but stable

Example:

- `child::node()[position()=2]` returns the second child
- `ancestor::node()[position()=2]` returns the grand-parent (the parent has position 1, the grand-parent position 2, and so on ...)

ABBREVIATIONS

XPath has an abbreviated syntax

- Child is the default axis, so `/child::rcp:recipe` is equivalent to `/rcp:recipe`
- The `attribute` axis may be replaced by the @ character
- The fragment `descendant-or-self::node()` may be replaced by //
- The fragment `self::node()` may be replaced by `.`
- The fragment `parent::node()` may be replaced by `..`

ABBREVIATIONS

XPath has an abbreviated syntax

- Child is the default axis, so `/child::rcp:recipe` is equivalent to `/rcp:recipe`
- The `attribute` axis may be replaced by the @ character
- The fragment `descendant-or-self::node()` may be replaced by //
- The fragment `self::node()` may be replaced by `.`
- The fragment `parent::node()` may be replaced by `..`

What is the unabbreviated syntax for the following expression?

```
//rcp:nutrition[@calories=349]/..//rcp:title/text()
```

BEYOND THE BASICS: GENERAL EXPRESSIONS

A general XPath expression evaluates to a sequence of items, where each item is either an node or an atomic value

Atomic values may be:

- numbers
- booleans
- Unicode strings
- or one of the other datatypes defined in XML Schema (see later)

And we already know that each node has an identity

GENERAL EXPRESSIONS: LITERAL EXPRESSIONS

Examples:

- `42`
- `3.1415`
- `6.022E13`
- `'XPath is a lot of fun'`
- `"XPath is a lot of fun"`
- `'The cat said "Meow!"'`
- `"The cat said ""Meow!"""`
- `"XPath is just
  so much fun"`

GENERAL EXPRESSIONS: ARITHMETIC EXPRESSIONS

Examples:

- `42 + 34`
- `42 - 34`
- `3.1415 * 5`
- `3.1415 div 5`
- `42 idiv 34`
- `42 mod 3`

Operators are generalized to sequences

- If any argument is empty, the result is empty
- If all argument are singleton sequences of numbers, the operation is performed
- Otherwise, a runtime error occurs

GENERAL EXPRESSIONS: VARIABLE REFERENCES

Examples:

- `$foo`
- `$bar:foo`

Variables are set in the context

- Can be set by the application before running the xpath expression (input variables)
- But variables can also be bound by the XPath for-expression (see later)

Careful!

- `$foo-17` refers to the variable “foo-17”
- To substract 17 from the value of variable “foo”, we write `($foo)-17`, `$foo -17`, or `$foo+-17`

Operators are generalized to sequences

- The ',' operator concatenates sequences
- Integer ranges are constructed with 'to'
- Sequences containing only nodes can be combined using the set operators `union`, `intersect`, `except`. The result is always sorted in document order, and duplicate-free.
- Sequences are always flattened

The following expressions all give the same result:

- `(1, (2,3,4), ((5)), (), (((6,7)), 8,9))`
- `1 to 9`
- `1,2,3,4,5,6,7,8,9`

GENERAL EXPRESSIONS: FUNCTIONS

- XPath has an extensive function library
- Default namespace for functions:
  `http://www.w3.org/2006/xpath-functions`
- 106 functions are required
- More functions with the namespace:
  `http://www.w3.org/2001/XMLSchema`
- See book for pages 78-81 for useful standard functions

Example

- calling a function with 4 arguments: `fn:avg(1,2,3,4)`
- calling a function with 1 argument: `fn:avg((1,2,3,4))`

GENERAL EXPRESSIONS: PATH EXPRESSIONS

- Location Paths are expressions
- They may start from arbitrary sequences that contain only nodes
  - evaluate the path for each node in the sequence
  - using the current node as context node
  - context position and size are taken from the sequence
  - the results are combined in document order

Example:

- `(fn:doc("veggie.xml"), fn:doc("bbq.xml"))//rcp:title`

(For multiple documents, the document order between nodes in different documents is implementation-dependent, but stable).

GENERAL EXPRESSIONS: FILTER EXPRESSIONS

- Predicates are generalized to arbitrary sequences
- The expression '.' is the context item

Example (what is the result?):

- `(10 to 40)[. mod 5 = 0 and position()>20]`

GENERAL EXPRESSIONS: VALUE COMPARISON

- Operators `eq`, `ne`, `lt`, `le`, `gt`, `ge`
- Used on atomic values
- when applied to an arbitrary values:
  - atomize
  - if either argument is empty, the result is empty
  - if either has length$>$ 1, raise runtime error (error in book)
  - if incomparible (e.g. string comparison with float), raise runtime error
  - otherwise, compare the two atomic values

Examples:

- `8 eq 4+4`
- `(//rcp:ingredient)[1]/@name eq "beef cube steak"`

ATOMIZATION

Definition

Atomization is the process of transforming an arbitrary sequence into a sequence of atomic values.

- Each node is replaced by its atomized value
- For element nodes, the atomized value is the concatenation of the content of all descendant text nodes (in document order)
- For other nodes, the atomized value is the obvious string

GENERAL EXPRESSIONS: GENERALIZED COMPARISON

- Operators `=`, `!=`, `<`, `<=`, `>`, `>=`
- Used on arbitrary values:
  - atomize
  - if either argument is empty, the result is empty
  - if there exists two atomic values, one from each atomized argument, whose comparison holds, the result is true
  - otherwise, the result is false
- So, this is an existential semantics!

Examples:

- `8 = 4+4`
- `(1,2) = (2,4)`
- `//rcp:ingredient/@name = "salt"`

GENERAL EXPRESSIONS: NODE COMPARISON

- Operators `is`, «, »,
- Used to compare nodes on identity and document order
- When applied to arbitrary values:
  - if either argument is empty, the result is empty
  - if both are singleton nodes, the nodes are compared
  - otherwise, a runtime error is raised

Examples:

- `(//rcp:recipe)[2] is //rcp:recipe[rcp:title eq "Ricotta Pie" ]`
- `/rcp:collection « (//rcp:recipe)[4]`
- `(//rcp:recipe)[4] » (//rcp:recipe)[3]`

BE CAREFUL ABOUT COMPARISONS

What is the result of the following comparisons?

- `((//rcp:ingredient)[40]/@name, (//rcp:ingredient)[40]/@amount) eq ((//rcp:ingredient)[53]/@name, (//rcp:ingredient)[53]/@amount)`
- `((//rcp:ingredient)[40]/@name, (//rcp:ingredient)[40]/@amount) = ((//rcp:ingredient)[53]/@name, (//rcp:ingredient)[53]/@amount)`
- `((//rcp:ingredient)[40]/@name, (//rcp:ingredient)[40]/@amount) is ((//rcp:ingredient)[53]/@name, (//rcp:ingredient)[53]/@amount)`

BE CAREFUL ABOUT COMPARISONS (2)

XPath violates most algebraic laws of equality

- Reflexivity:
  `()=()` yields `false`
- Transitivity:
  `(1,2)=(2,3)` and `(2,3)=(3,4)`, but not `(1,2)=(3,4)`
- Anti-symmetry:
  `(1,4)<=(2,3)` and `(2,3)<=(1,4)` but not `(1,2)=(3,4)`
- Negation:
  both `(1)!=()` and `(1)=()` yield `false`

GENERAL EXPRESSIONS: BOOLEAN EXPRESSIONS

- Operators `and`, `or`, `not`,
- Arguments are interpreted as `false` if
  - the boolean false
  - the empty sequence
  - the empty string
  - the number zero
- Constants using functions `true()` and `false()`

Example:

- `(//rcp:recipe) and //ingredient[4]/@quantity`

GENERAL EXPRESSIONS: FOR EXPRESSIONS

General form `for $`*var* `in` *expr*$_1$ `return` *expr*$_2$

- evaluates *expr*$_1$ to a sequence
- for each item in that sequence evaluate *expr*$_2$ with *var* bound to the current item
- concatenates the resulting sequences

Examples:

- `for $r in //rcp:recipe return fn:count($r//rcp:ingredient[fn:not(rcp:ingredient)])`
- ```
  for $i in (1 to 5)
    for $j in (1 to $i)
      return $j
  ```
  ```

Example:

- ```
  fn:avg(
    for $r in //rcp:ingredient return
      if ( $r/@unit = "cup" )
        then xs:double($r/@amount) * 237
      else if ( $r/@unit = "teaspoon" )
        then xs:double($r/@amount) * 5
      else if ( $r/@unit = "tablespoon" )
        then xs:double($r/@amount) * 15
      else ()
    )
  ```
  ```

GENERAL EXPRESSIONS: QUANTIFIED EXPRESSIONS

Example:

- `some $r in //rcp:ingredient satisfies $r/@name eq "sugar"`
- `every $r in //rcp:recipy satisfies $r/@name neq "cake"`

XPath 1.0 restrictions

- Many implementations only support XPath 1.0
- Smaller function library
- Implicit casts of values
- Some expressions change semantics:

  `"4" < "4.0"`

  is false in XPath 1.0 but true in XPath 2.0

XPath 3.0 extensions

New features in XPath 3.0 include:

- String concatenation operator `||`
- Simple mapping operator `!`. E_1 `!` E_2 is similar to E_1/E_2 except that:
  - E_1 can return an arbitrary sequence, not necessarily a sequence of nodes
  - E_1 is evaluated to a sequence. For every item in that sequence E_2 is evaluated with the current item as context item.
  - Results are concatenated, but there is no duplicate elimination, nor sorting after the concatenation.
- Higher-order functions: functions are valid items; anonymous functions can be defined and called dynamically.
- Let binding.

Examples:

- `'ab' || 'def'` = `'abdef'`
- ```
  let $x := doc('a.xml')/*, $y := $x//*
  return $y[@value gt $x/@min]
  ```
  ```

XPATH 3.1 EXTENSIONS

XPath 3.1 extends the XPath Data Model

- A value in xpath 3.1 is a sequence of 0 or more items, where every item is an atomic value, a node, a map, an array, or a function.
- Maps are dictionaries from keys to values.
  - Keys must be atomic values.
  - Duplicate keys are not allowed.
- Arrays are Arrays are ordered collections of items (like sequences), but an array can include as element a sequence (which a sequence cannot).
- New expressions to create and manipulate maps and arrays.
- This allows to also load json.

Create a map with 3 entries:

- `map { "Su" : "Sunday", "Mo" : "Monday", "Tu" : "Tuesday"}`

XPATH 3.1 EXTENSIONS

XPath 3.1 extends the XPath Data Model

- A value in xpath 3.1 is a sequence of 0 or more items, where every item is an atomic value, a node, a map, an array, or a function.
- Maps are dictionaries from keys to values.
  - Keys must be atomic values.
  - Duplicate keys are not allowed.
- Arrays are Arrays are ordered collections of items (like sequences), but an array can include as element a sequence (which a sequence cannot).
- New expressions to create and manipulate maps and arrays.
- This allows to also load json.

Maps can be nested:

- `map { "book" : map { "author" : ["J. Doe", "X. Brown"], "title": "The X"}}`

XPATH 3.1 EXTENSIONS

XPath 3.1 extends the XPath Data Model

- A value in xpath 3.1 is a sequence of 0 or more items, where every item is an atomic value, a node, a map, an array, or a function.
- Maps are dictionaries from keys to values.
  - Keys must be atomic values.
  - Duplicate keys are not allowed.
- Arrays are Arrays are ordered collections of items (like sequences), but an array can include as element a sequence (which a sequence cannot).
- New expressions to create and manipulate maps and arrays.
- This allows to also load json.

Maps lookup

- `$x("book")`

XPATH 3.1 EXTENSIONS

XPath 3.1 extends the XPath Data Model

- A value in xpath 3.1 is a sequence of 0 or more items, where every item is an atomic value, a node, a map, an array, or a function.
- Maps are dictionaries from keys to values.
  - Keys must be atomic values.
  - Duplicate keys are not allowed.
- Arrays are Arrays are ordered collections of items (like sequences), but an array can include as element a sequence (which a sequence cannot).
- New expressions to create and manipulate maps and arrays.
- This allows to also load json.

Array creation

- Array with three members: `[1, 2, "book"]`
- Array with two members, both sequences: `[ (), (27, 17, 0)]`
- Array with two members, the second being a sequence: `[1, $x//recipe]`
- Curly array constructor creates flat arrays: `array {1, $x//recipe}`

JSON

JSON, THAT OTHER XML

JSON = JavaScript Object Notation

- A syntactical fragment of JavaScript (aka EcmaScript) for describing data
- Libraries for reading & writing JSON exist for virtually every programming language
- Widely used with JavaScript & AJAX

Example:

```
{ "company": { "name": "WIND solutions",  "symbol": "WIND" },
  "reportData":[
    { "year": 2011, "profit": 2000000 },
    { "year": 2010, "deficit": 1000   },
  ],
  "author": "John Doe"
}
```

- JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- A value is a a string, number, boolean, null, object, or array.

Number primitive type

- A number is any integer or fractional number like in most programming languages

Example:

```
10      -12.34      10e5      +3.141516
```

- JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- A value is a a string, number, boolean, null, object, or array.

String primitive type

- A string is a sequence of zero or more unicode characters.
- String literals are delimited by double quotes. The C-style escape characters (\", \n, \t, ...) apply.

Example:

"This is a string literal\n This continues on a new line"

- JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- A value is a a string, number, boolean, null, object, or array.

Boolean primitive types

- A boolean is either the literal true or false

JSON IN A NUTSHELL

- JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- A value is a a string, number, boolean, null, object, or array.

Object structured type

- An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.
- Objects are delimited by curly braces; within the same object, each name must be unique.

Example:

```
{ "company": { "name": "WIND solutions",  "symbol": "WIND" },
  "reportData":[
    { "year": 2011, "profit": 2000000 },
    { "year": 2010, "deficit": 1000   },
  ],
  "author": "John Doe"
}
```

- JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).
- A value is a a string, number, boolean, null, object, or array.

Array structured type

- An array is an ordered sequence of zero or more values.
- Arrays are delimited by square brackets.

Example:

```
[
    { "year": 2011, "profit": 2000000 },
    { "year": 2010, "deficit": 1000   },
]
```