Xquery, DTD, and XML Schema

Rasmus Pagh

Today's lecture

- XML tools, part 2:
  - Xquery
  - Schema languages:
    - DTD
    - XML Schema

An Introduction to XML and Web Technologies

Querying XML Documents with XQuery

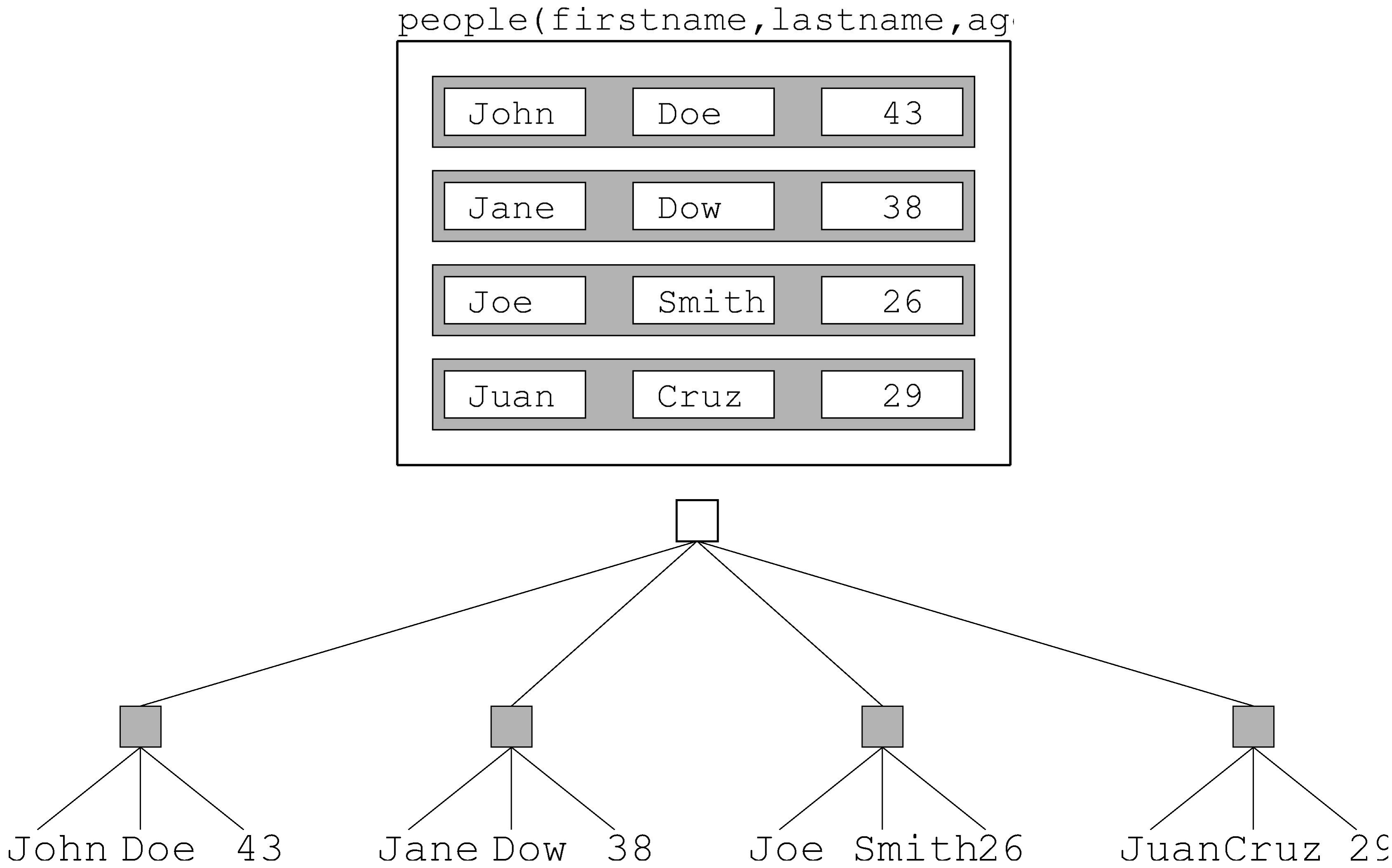
following slides based on slides by

Anders Møller & Michael I. Schwartzbach

XQuery 1.0

- XML documents naturally generalize database relations
- XQuery is the corresponding generalization of SQL

From Relations to Trees



Trees Are Not Relations

- Not all trees satisfy the previous characterization
- Also, XML trees are *ordered*, while both rows and columns of tables may be permuted without changing the meaning of the data

Relationship to XPath

- XQuery 1.0 is a *strict superset* of XPath 2.0
- Every XPath 2.0 expression is directly an XQuery 1.0 expression (a query)
- The extra expressive power is the ability to
  - *join* information from different sources and
  - *generate* new XML fragments
- Main construct: FLWOR expression
  - conceptually similar to select-from-where
  - syntax similar to imperative language

FLWOR Example

```
<doubles>
  { for $s in fn:doc("students.xml")//student
    let $m := $s/major
    where fn:count($m) ge 2
    order by $s/@id
    return <double>
             { $s/name/text() }
           </double>
  }
</doubles>
```

XML Expressions

- XQuery expressions may compute *new XML nodes*
- Expressions may denote element, character data, comment, and processing instruction nodes
- Each node is created with a unique *node identity*

Direct Constructors

- Uses the standard XML syntax
- The expression

```
<foo><bar/>baz</foo>
```

  evaluates to the given XML fragment
- Identity:

```
<foo/> is <foo/>
```

  evaluates to false

The Difference Between For and Let (1/4)

```
for $x in (1, 2, 3, 4)
let $y := ("a", "b", "c")
return ($x, $y)
```

```
1, a, b, c, 2, a, b, c, 3, a, b, c, 4, a, b, c
```

The Difference Between For and Let (2/4)

```
let $x := (1, 2, 3, 4)
for $y in ("a", "b", "c")
return ($x, $y)
```

↓

```
1, 2, 3, 4, a, 1, 2, 3, 4, b, 1, 2, 3, 4, c
```

The Difference Between For and Let (3/4)

```
for $x in (1, 2, 3, 4)
for $y in ("a", "b", "c")
return ($x, $y)
```

↓

```
1, a, 1, b, 1, c, 2, a, 2, b, 2, c,
3, a, 3, b, 3, c, 4, a, 4, b, 4, c
```

The Difference Between For and Let (4/4)

```
let $x := (1, 2, 3, 4)
let $y := ("a", "b", "c")
return ($x, $y)
```

```
1, 2, 3, 4, a, b, c
```

Computing Joins

- Join is implemented as nested loops
  - But not necessarily executed that way!

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
for $r in fn:doc("recipes.xml")//rcp:recipe
for $i in $r//rcp:ingredient/@name
for $s in fn:doc("fridge.xml")//stuff[text()=$i]
return $r/rcp:title/text()
```

```
<fridge>
  <stuff>eggs</stuff>
  <stuff>olive oil</stuff>
  <stuff>ketchup</stuff>
  <stuff>unrecognizable moldy thing</stuff>
</fridge>
```

Example: Inverting a Relation

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
<ingredients>
  { for $i in distinct-values(
                fn:doc("recipes.xml")//rcp:ingredient/@name)
    order by $i
    return <ingredient name="{$i}">
             { for $r in fn:doc("recipes.xml")//rcp:recipe
               where $r//rcp:ingredient[@name=$i]
               return <title>{$r/rcp:title/text()}</title>
             }
           </ingredient>
  }
</ingredients>
```

Semantics of FLWOR

- `let $a := <e>`: Assign a value to local variable `$a`, given by expression `<e>`.
- `for $t in <e> <b>`: Iterate through the list given by `<e>`, binding `$t` to each item and executing `<b>` to build output list.
- `where <p>`: If predicate `<p>` is not satisfied, go to next binding in `for`.
- `return <e>`: Add `<e>` to output list of enclosing expression.
- `order by $b`: Order output list by `$b`.

Summary

- XML trees generalize relational tables
- XQuery similarly generalizes SQL

- Next week: XSLT
  - XQuery and XSLT have roughly the same expressive power
  - Suited for different application domains:
    - Xquery is created for querying
    - XSLT is targeted at presentation/transformation

An Introduction to XML and Web Technologies

Schema Languages

following slides based on slides by
Anders Møller & Michael I. Schwartzbach

Next

- The **purpose** of using schemas
- The schema languages **DTD** and **XML Schema**
- **Regular expressions** – a commonly used formalism in schema languages

Motivation

- We have seen a Recipe Markup Language ...but so far only **informally** described its **syntax**

- *How can we make tools that check that an XML document is a* ***syntactically correct*** *Recipe Markup Language document (and thus meaningful)?*

- Implementing a specialized validation tool for Recipe Markup Language is *not* the solution...

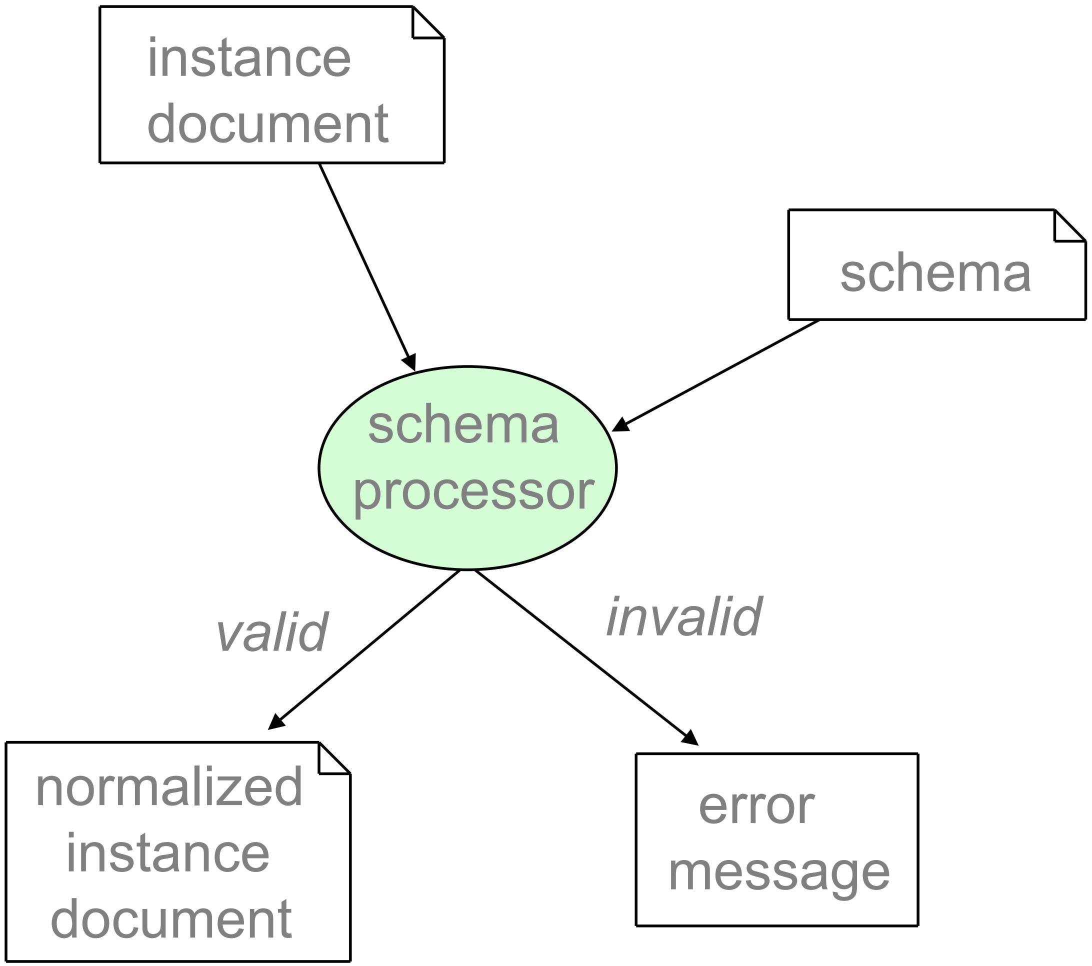
XML Languages

- ***XML language***:
  a set of XML documents with some semantics

- ***schema***:
  a formal definition of the *syntax* of an XML language (not its semantics)

- ***schema language***:
  a notation for writing schemas

Validation



instance document
schema
schema processor
valid
invalid
normalized instance document
error message

Why use Schemas?

- Formal but human-readable descriptions
  - basis for writing programs that read files from the markup language

- Data validation can be performed with existing schema processors

Regular Expressions

- Commonly used in schema languages to describe **sequences** of **characters** or **elements**
- Σ: an alphabet (e.g Unicode characters or element names)
- Regular expressions are recursively defined:
  - σ matches the character $\sigma \in \Sigma$
  - α? matches zero or one α
  - α* matches zero or more α's
  - α+ matches one or more α's
  - $\alpha\ \beta$ matches any concatenation of an α and a β
  - $\alpha \mid \beta$ matches the union of α and β

Examples

- A regular expression describing **integers**:

```
0|-?(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
```

- A regular expression describing the valid contents of **`table`** elements in XHTML:

```
caption? (col*|colgroup*) thead? tfoot? (tbody+ | tr+)
```

DTD – Document Type Definition

- Specified as an integral part of XML 1.0
- A starting point for development of more expressive schema languages
- Considers elements, attributes, and character data – processing instructions and comments are mostly ignored

Document Type Declarations

- Associates a DTD schema with the instance document
- Example:

```
<?xml version="1.1"?>
<!DOCTYPE collection SYSTEM "http://www.brics.dk/ixwt/
recipes.dtd">
<collection>
...
</collection>
```

Element Declarations

<!ELEMENT *element-name* *content-model* >

Content models:

- **EMPTY**
- **ANY**
- ***mixed content***: (#PCDATA|e_1|e_2|...|e_n)*
- ***element content***: regular expression over element names (concatenation is written with ",")

Example:

```
<!ELEMENT table
        (caption?,(col*|colgroup*),thead?,tfoot?,(tbody+|tr+)) >
```

Attribute-List Declarations

<!ATTLIST *element-name attribute-definitions* >

Each attribute definition consists of

- an attribute name
- an attribute *type*
- a *default declaration*

Example:

```
<!ATTLIST input maxlength CDATA #IMPLIED
                tabindex CDATA #IMPLIED>
```

Attribute Types

- CDATA: any value
- *enumeration*: $(s_1|s_2|...|s_n)$
- ID: must have unique value
- IDREF (/ IDREFS): must match some ID attribute(s)
- ...

Examples:

```
<!ATTLIST p align (left|center|right|justify)
#IMPLIED>

<!ATTLIST recipe id ID #IMPLIED>
<!ATTLIST related ref IDREF #IMPLIED>
```

Attribute Default Declarations

- #REQUIRED
- #IMPLIED (= optional)
- ”*value*” (= optional, but default provided)
- #FIXED ”*value*” (= required, must have this value)

Example:

```
<!ATTLIST form
      action CDATA #REQUIRED
      onsubmit CDATA #IMPLIED
      method (get|post) "get"
      enctype CDATA "application/x-www-form-urlencoded" >
```

RecipeML with DTD (1/2)

```
<!ELEMENT collection (description,recipe*)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT recipe
  (title,date,ingredient*,preparation,comment?,
   nutrition,related*)>
<!ATTLIST recipe id ID #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>
```

RecipeML with DTD (2/2)

```
<!ELEMENT preparation (step*)>
<!ELEMENT step (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition calories CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    protein CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>
<!ELEMENT related EMPTY>
<!ATTLIST related ref IDREF #REQUIRED>
```

Some limitations of DTD

1. Cannot constrain **character data**
2. Specification of **attribute values** is too limited
3. **Character data** cannot be combined with the **regular expression** content model
4. The support for **modularity**, **reuse**, and **evolution** is too primitive
5. No support for **namespaces**

XML Schema is a newer schema language with fewer limitations.

XML Schema example (1/3)

Instance document:

```
<b:card xmlns:b="http://businesscard.org">
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.com</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo b:uri="widget.gif"/>
</b:card>
```

XML Schema example (2/3)

Schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  <element name="name" type="string"/>
  <element name="title" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
  <element name="logo" type="b:logo_type"/>
  <attribute name="uri" type="anyURI"/>
```

XML Schema example (3/3)

```
  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      <element ref="b:title"/>
      <element ref="b:email"/>
      <element ref="b:phone" minOccurs="0"/>
      <element ref="b:logo" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="logo_type">
    <attribute ref="b:uri" use="required"/>
  </complexType>
</schema>
```

XML Schema Types and Declarations

- **Simple type definition**:
  defines a family of Unicode text strings
- **Complex type definition**:
  defines a content and attribute model
- **Element declaration**:
  associates an element name with a simple or complex type
- **Attribute declaration**:
  associates an attribute name with a simple type

Element and Attribute Declarations

Examples:

```
– <element name="serialnumber"
               type="nonNegativeInteger"/>

– <attribute name=”alcohol"
                type=”r:percentage"/>
```

Derived simple types

```
<simpleType name="score_from_0_to_100">
  <restriction base="integer">
    <minInclusive value="0"/>
    <maxInclusive value="100"/>
  </restriction>
</simpleType>

<simpleType name="percentage">
  <restriction base="string">
    <pattern value="([0-9]|[1-9][0-9]|
  100)%"/>
  </restriction>
</simpleType>
```

regular expression

Simple Type Derivation – Union

```
<simpleType name="boolean_or_decimal">
  <union>
    <simpleType>
      <restriction base="boolean"/>
    </simpleType>
    <simpleType>
      <restriction base="decimal"/>
    </simpleType>
  </union>
</simpleType>
```

Complex Types

- Content models as **regular expressions**:
  - Element reference <element ref=”*name*”/>
  - Concatenation <sequence> ... </sequence>
  - Union <choice> ... </choice>
  - All <all> ... </all>
  - Element wildcard: <any namespace=”*...*” processContents=”*...*”/>
- Attribute reference: <attribute ref=”*...*”/>
- Attribute wildcard: <anyAttribute namespace=”*...*” processContents=”*...*”/>

Cardinalities: minOccurs, maxOccurs, use=“required”

Mixed content: mixed=”true”

Example

```
<element name="order" type="n:order_type"/>

<complexType name="order_type" mixed="true">
  <choice>
    <element ref="n:address"/>
    <sequence>
      <element ref="n:email"
               minOccurs="0"
  maxOccurs="unbounded"/>
      <element ref="n:phone"/>
    </sequence>
  </choice>
  <attribute ref=”n:id" use="required"/>
</complexType>
```

Global vs. Local Descriptions

Global (toplevel) style:

```
<element name="card“
         type="b:card_type"/>
<element name="name“
         type="string"/>

<complexType name="card_type">
  <sequence>
    <element ref="b:name"/>
    ...
  </sequence>
</complexType>
```

Local (inlined) style:

```
<element name="card">
  <complexType>
    <sequence>
      <element name="name"
               type="string"/>
      ...
    </sequence>
  </complexType>
</element>
```

inlined

Requirements to Complex Types

- **Two element declarations** that have the **same name** and appear **in the same complex type** must have **identical types**

```
<complexType name="some_type">
  <choice>
    <element name="foo" type="string"/>
    <element name="foo" type="integer"/>
  </choice>
</complexType>
```

  - This requirement makes efficient implementation easier

- `all` can only contain `element` (e.g. not `sequence`)

Uniqueness, Keys, References

```
<element name="w:widget" xmlns:w="http://www.widget.org">
  <complexType>
  ...
  </complexType>
  <key name="my_widget_key">
    <selector xpath="w:components/w:part"/>
    <field xpath="@manufacturer"/>
    <field xpath="w:info/@productid"/>
  </key>
  <keyref name="annotation_references"
   refer="w:my_widget_key">
    <selector xpath=".//w:annotation"/>
    <field xpath="@manu"/>
    <field xpath="@prod"/>
  </keyref>
</element>
```

in every widget, each part must have unique (manufacturer, productid)

only a “downward” subset of XPath is used

in every widget, for each annotation, (manu, prod) must match a my_widget_key

unique: as key, but fields may be absent

RecipeML with XML Schema (1/5)

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.brics.dk/ixwt/recipes"
        targetNamespace="http://www.brics.dk/ixwt/recipes"
        elementFormDefault="qualified">

  <element name="collection">
    <complexType>
      <sequence>
        <element name="description" type="string"/>
        <element ref="r:recipe" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
    <unique name="recipe-id-uniqueness">
      <selector xpath=".//r:recipe"/>
      <field xpath="@id"/>
    </unique>
    <keyref name="recipe-references" refer="r:recipe-id-uniqueness">
      <selector xpath=".//r:related"/>
      <field xpath="@ref"/>
    </keyref>
  </element>
```

RecipeML with XML Schema (2/5)

```
<element name="recipe">
  <complexType>
    <sequence>
      <element name="title" type="string"/>
      <element name="date" type="string"/>
      <element ref="r:ingredient" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="r:preparation"/>
      <element name="comment" type="string" minOccurs="0"/>
      <element ref="r:nutrition"/>
      <element ref="r:related" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="id" type="NMTOKEN"/>
  </complexType>
</element>
```

RecipeML with XML Schema (3/5)

```
<element name="ingredient">
  <complexType>
    <sequence minOccurs="0">
      <element ref="r:ingredient" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="r:preparation"/>
    </sequence>
    <attribute name="name" use="required"/>
    <attribute name="amount" use="optional">
      <simpleType>
        <union>
          <simpleType>
            <restriction base="r:nonNegativeDecimal"/>
          </simpleType>
          <simpleType>
            <restriction base="string">
              <enumeration value="*"/>
            </restriction>
          </simpleType>
        </union>
      </simpleType>
    </attribute>
    <attribute name="unit" use="optional"/>
  </complexType>
</element>
```

RecipeML with XML Schema (4/5)

```
<element name="preparation">
  <complexType>
    <sequence>
      <element name="step" type="string“ minOccurs="0“ maxOccurs="unbounded"/
 >
    </sequence>
  </complexType>
</element>

<element name="nutrition">
  <complexType>
    <attribute name="calories" type="r:nonNegativeDecimal“ use="required"/>
    <attribute name="protein" type="r:percentage" use="required"/>
    <attribute name="carbohydrates" type="r:percentage" use="required"/>
    <attribute name="fat" type="r:percentage" use="required"/>
    <attribute name="alcohol" type="r:percentage" use="optional"/>
  </complexType>
</element>

<element name="related">
  <complexType>
    <attribute name="ref" type="NMTOKEN" use="required"/>
  </complexType>
</element>
```

RecipeML with XML Schema (5/5)

```
  <simpleType name="nonNegativeDecimal">
    <restriction base="decimal">
      <minInclusive value="0"/>
    </restriction>
  </simpleType>

  <simpleType name="percentage">
    <restriction base="string">
      <pattern value="([0-9]|[1-9][0-9]|100)%"/>
    </restriction>
  </simpleType>

</schema>
```

Strengths of XML Schema

- Namespace support
- Data types (built-in and derivation)
- Modularization
- Type derivation mechanism

RELAX NG

- OASIS + ISO competitor to XML Schema
- Designed for simplicity and expressiveness, solid mathematical foundation
- Several other proposals, e.g. DSD2.

Summary

- **Schema**: formal description of the syntax of an XML language
- **DTD**: simple schema language
  - elements, attributes, entities, ...
- **XML Schema**: more advanced schema language
  - element/attribute declarations
  - simple types, complex types, type derivations
  - global vs. local descriptions
  - ...

Next weeks

- Only two 1-hour sessions left!
  - XSLT
  - Exam run-through (preparation: 4 hours)
- Three possibilities:
  - A) 8-10 AM next week
  - B) 8-10 AM in two weeks
  - C) 9-10 AM next week and in two weeks

- Vote: What do you prefer?

More XML Schema

- The following slides give more information and examples on XML Schema.
- They are part of the course curriculum and can be considered supplements to the course literature.

Simple Types – Primitive

`string`	*any Unicode string*
`boolean`	`true`, `false`, `1`, `0`
`decimal`	`3.1415`
`float`	`6.02214199E23`
`double`	`42E970`
`dateTime`	`2004-09-26T16:29:00-05:00`
`time`	`16:29:00-05:00`
`date`	`2004-09-26`
`hexBinary`	`48656c6c6f0a`
`base64Binary`	`SGVsbG8K`
`anyURI`	`http://www.brics.dk/ixwt/`
`QName`	`rcp:recipe`, `recipe`
...	

Simple Type Derivation – List

```
<simpleType name="integerList">
  <list itemType="integer"/>
</simpleType>
```

matches whitespace separated lists of `integers`