Querying and Storing XML

Week 3
XSLT, DTDs, Schemas, Constraints
January 29-Feb 1, 2013

XSLT

- XML StyLesheet Language Transformations
- Goal: Transform XML to other formats
  - HTML
  - other XML languages
  - text
  - PDF, formatting (XSL:FO)
- Mainly aimed at generating/transforming "XML documents", not querying "XML data"

Basic idea

- A stylesheet is a collection of *rules*
- Each rule specifies a *selector* and an *action*

```
<xsl:template match="xpath">
  action
</xsl:template>
```

  - The selector defines when the rule applies
    - If more than one rule applies, use "most specific"
  - The action defines the result produced by the rule

Simple example

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
  <xsl:output method="text"/>
  <xsl:template match="*">
   Hello world!
  </xsl:template>
</xsl:stylesheet>
```

- Selects any node
- Replaces with "Hello world!"
- **Doesn't recurse - just ignores rest of document**

Example (1)

- Say we have some “XML data”:

```
<records>
  <record>
    <a>1</a>
    <b>2</b>
  </record>
  ...
</records>
```


Example (2)

- Start at root, generate HTML boilerplate

```
<xsl:output method="html"/>
<xsl:template match="/">
 <html>
  <head><title>Example</title></head>
  <body>
    <table frame="box" rules="all">
     <tr><th> A </th><th> B </th> </tr>
      <xsl:apply-templates/>
    </table>
  </body>
 </html>
</xsl:template>
```


Example (3)

- Replace “records” with table rows

```
<xsl:template match="record">
 <tr>
  <td><xsl:copy-of select="a/text()"/></td>
  <td><xsl:copy-of select="b/text()"/></td>
 </tr>
</xsl:template>
```


More XSLT

- Can use *names* and *modes* to organize templates
  - Define subsets of rules callable/applicable by name
- Allows variables & parameters
- Can generate unique IDs for nodes
- Essentially a full-featured programming language
  - Turing-complete
  - but some things still “hard”, e.g. XQuery-style joins
- More examples online, e.g iTunes album listing
  - http://www.movable-type.co.uk/scripts/itunes-albumlist.html

DTDs

Types and XML

- XML stands for *eXtensible Markup Language*
- Extensibility means you can define your own markup languages
  - via types, or *schemas*
- *Well-formed*: just means the opening & closing tags match etc.
- *Valid*: means there is a schema and the document matches it

Goals of typing

- Interoperability/reliability
  - specify required, optional, default values
- Consistency
  - ensure updates or generated output is "sensible"
- Efficiency
  - use to organize storage
  - use as basis for query optimization

Schemas

- Many schema languages/formalisms have been considered
  - DTDs (XML 1.0)
  - XML Schema (W3C)
  - Relax/NG (OASIS), DSD, Schematron, ...
  - Regular expression types (XDuce, XQuery)
- Most of these are based on regular expressions in some way

Document type definitions (DTDs)

- Came with XML 1.0
- Element declarations <!ELEMENT elt (content)>
  - declare elt to have content content
  - content usually a regular expression over element names
  - also allowed: ANY, EMPTY, PCDATA (text)
- Attribute declarations <!ATTLIST elt att ...>
  - declare elt to have an attribute named att
  - where att's type is CDATA (text)
  - other types possible, more later

Regular expressions

- Recall simple regular expressions
- r ::= ∅ | ε | a | rs | r + s | r*
- ∅ - empty set
- ϵ - empty sequence
- a - single symbol "a" (in DTD, an elt name)
- rs - sequential composition {xy | x ϵ r, y ϵ s }
- r+s - union
- r* - iteration {$x_1..x_n$ | x_i ϵ r}

Regular expressions

- Recall simple regular expressions
- r ::= ∅ | ε | a | rs | r + s | r*
- ∅ - empty set
- ϵ - empty sequence
- a - single symbol "a" (
- rs - sequential compositio
- r+s - union
- r* - iteration {$x_1..x_n$ | x_i ϵ r}

Written r | s in DTDs

Regular expressions

- Recall simple regular expressions
- r ::= ∅ | ε | a | rs | r + s | r*
- ∅ - empty set
- ϵ - empty sequence
- a - single symbol "a" (
- rs - sequential compositio
- r+s - union
- r* - iteration {$x_1..x_n$ | x_i ϵ r}

Definable:
r+ = r,r*
r? = r+ε

Example

```
<!ELEMENT root (row*)>
<!ATTLIST root title CDATA #REQUIRED>
<!ELEMENT row (A,(B|C))>
<!ATTLIST row id CDATA #REQUIRED>
<!ELEMENT A (#PCDATA)>
<!ELEMENT B (#PCDATA)>
<!ELEMENT C (#PCDATA)>
```

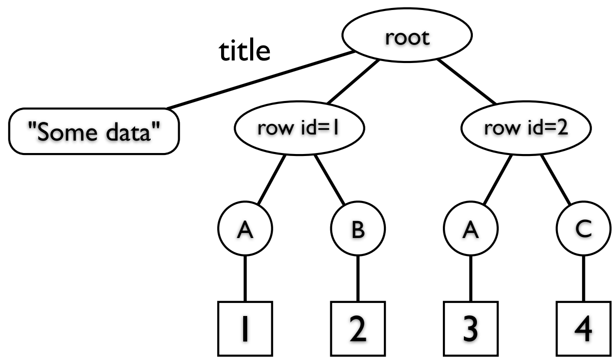

Alternative presentation (ignoring attributes)

```
root → row*
row → A,(B|C)
A → PCDATA
B → PCDATA
C → PCDATA
```

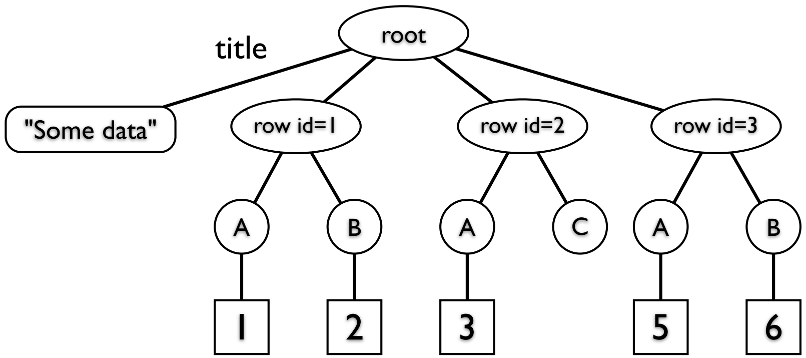

Attributes

- Attributes can be required (#REQUIRED), optional (#IMPLIED), fixed (#FIXED), or have a specified default value
- Attribute declarations can specify other types including:
- ID: Attribute value **must be unique** within document
  - <!ATTLIST book isbn ID #REQUIRED>
- IDREF: Attribute **must refer to an ID** elsewhere in document
  - <!ATTLIST book previous_isbn IDREF #IMPLIED>
- IDREFS: a list of multiple IDs
- Enumerations: one of a list
  - <!ATTLIST book type (comic|novel|textbook) #REQUIRED>

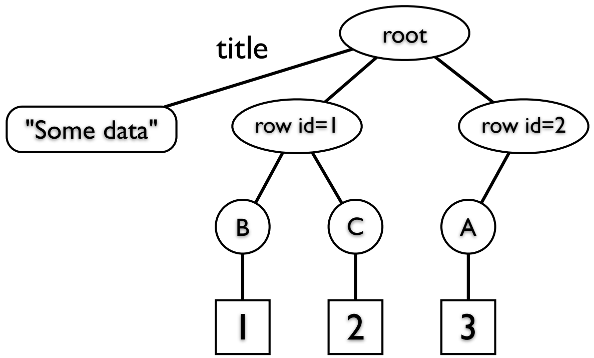
Valid



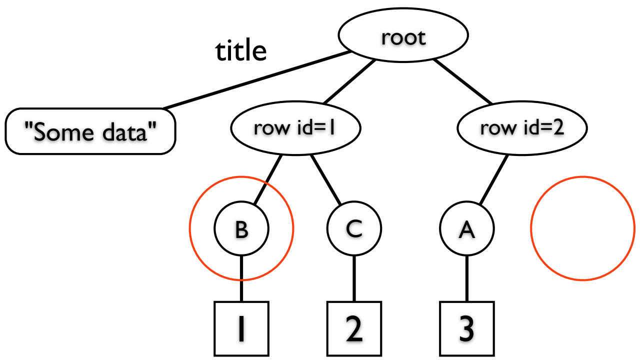
Still valid



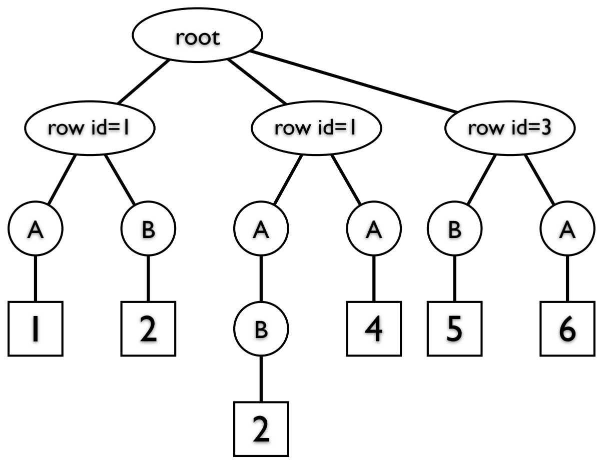
Invalid



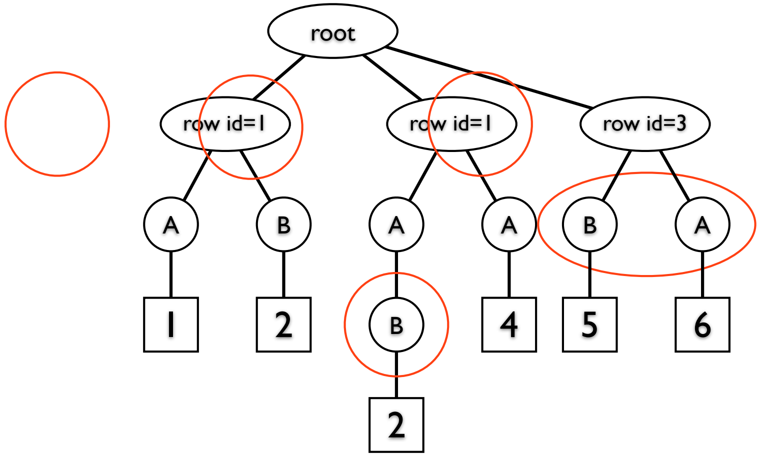
Invalid



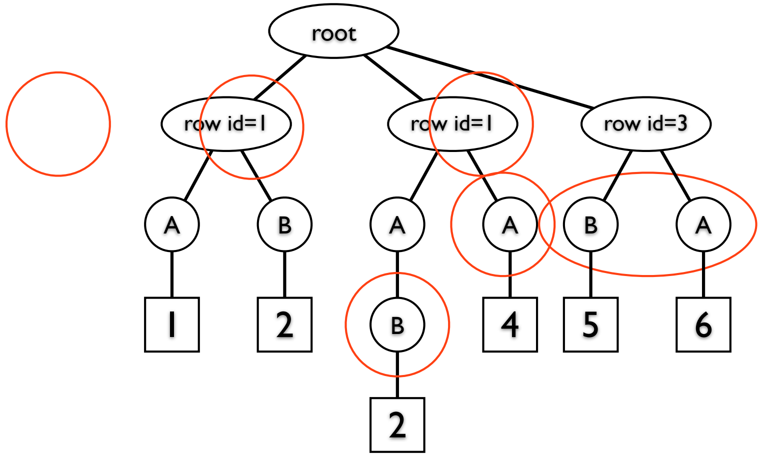
Quiz



Quiz



Quiz



Example: bibliography

```
bib -> (book|article)*
book -> author*, title, publisher, year
article -> author*, title, journal, volume, pages, year
author -> first, middle?, last
first -> PCDATA
middle -> PCDATA
last -> PCDATA
...
```


Example: syntax trees

```
exp -> plus|minus|times|div|num
num -> PCDATA
plus -> exp,exp+
minus -> exp,exp
times -> exp,exp+
div -> exp,exp
```

Restrictions on DTD

- Regular expressions must be "deterministic"
- Example: ((a|b),(a|c)) **not** determinstic
  - can't decide whether "a" in input matches first or second "a"
- However, equivalent to (a,(a|c))|(b,(a|c)), which is deterministic
- [Brueggemann-Klein & Wood, Inf. Comput. 229-253 (1998)]

Checking validity

- Traverse document
  - check that each element's actual children match specified regular expression
  - Check attribute types
  - Check ids are unique and idrefs refer to ids
- Can be done more efficiently
  - e.g. with one streaming pre-order pass over document
  - compiling regular expressions to automata

Recursive DTDs

- DTD rules can be recursive
  - `node → (node,node)?`
- Recursion increases complexity of DTD
  - This leads to documents of unbounded depth
  - Some element types might not have any finite matching trees
  - but this is easy to detect (look for unguarded cycles)
    - `silly → (silly, silly)`

Limitations

- Can't constrain text / attribute content (except in very limited ways)
- Element, attribute content **context insensitive**
  - can't use same tag, e.g. "name", in different ways
- Interleaving/unordered content not well supported
- ID/IDREF too simplistic (lack of typing, scope)

Next time

- XML Schemas
- Constraints on XML documents: Keys for XML

XML Schemas & Constraints

XML Schemas

- W3C standard, intended as replacement for DTDs
  - Namespaces
  - Built-in & defined types besides plain strings
  - Context sensitive typing/reuse of elt and att names
  - Numerical cardinality constraints, interleaving
  - Keys/uniqueness constraints

Namespaces

- XSLT, XML Schema are dialects of XML
  - XSLT can contain tags from other dialects
  - XML Schema can refer to other dialects when defining new one
- Namespace mechanism used to allow use of same element tag name in different contexts
  - `xsl:element` vs `xs:element`
- Not a major issue for "XML as data"
  - but needed to author XSLT, XML Schema documents

Simple types

- Simple types are (subsets of) strings
  - `string`
  - `boolean (true, false, 0, 1)`
  - `decimal, float, double`
  - `duration, time, date, dateTime, ..`
  - `hexBinary, base64Binary`
  - `anyURI, QName` (qualified name)
- Can define new simple types by restricting, forming lists, or taking unions of existing types

Complex types

- Describe possible element content
  - simple content, or
  - allowed attributes (must have simple types)
  - regular expression describing subelement structure
    - sequence (,), choice (|), any as in DTD
    - references to other elements, or **inline** declarations
    - **cardinality constraints** (minOccurs, maxOccurs) , generalizing regexp *, +, ?
    - **interleaving**/shuffling (sometimes written &)
  - subject to determinacy & other **restrictions**
  - can refer to named **groups** of elements

Example

```
<complexType name="DTDExample">
 <sequence>
  <element ref="A"/>
  <choice>
   <element ref="B"/>
   <element ref="C"/>
  </choice>
 </sequence>
</complexType>

<element name="A" simpleType="string"/>
<element name="B" simpleType="decimal"/>
<element name="C" simpleType="dateTime"/>
```

$\cong$ (A,(B|C))
but with different types
for content of A,B,C

Example: inlined version

```
<complexType name="DTDExample">
 <sequence>
  <element name="A" simpleType="string"/>
  <choice>
   <element name="B" simpleType="decimal"/>
   <element name="C" simpleType="dateTime"/>
  </choice>
 </sequence>
</complexType>
```

Example: inlined version

```
<complexType name="DTDExample">
 <sequence>
  <element name="A" simpleType="string"/>
  <choice>
   <element name="B" simpleType="decimal"/>
   <element name="C" simpleType="dateTime"/>
  </choice>
 </sequence>
</complexType>
```

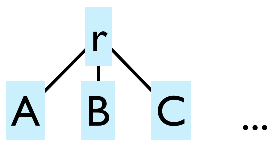
Element type information can be given inline (but then cannot be shared)

Cardinality constraints

- Most tags can have "`minOccurs`" and "`maxOccurs`" constraints; default `minOccurs` = `maxOccurs` = 1
  - `minOccurs` = 0 to simulate ?
  - `minOccurs` = unbounded to simulate +
  - `minOccurs` = 0, `maxOccurs` = unbounded to simulate *
- Can always be simulated using regexps but causes blowup in size of expression
  - consider `minOccurs` = 10, `maxOccurs` = 20
  - vs. (a,a,a,a,a,a,a,a,a,a,a?,a?,a?,a?,a?,a?,a?,a?,a?,a?)
  - (note this regexp is not determinstic either...)

Unordered content

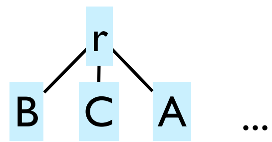
```
<all>
 <element ref="A"/>
 <element ref="B"/>
 <element ref="C"/>
</all>
```



- All of A,B,C **must** appear, but can be in **any order**
- Restriction: Only distinct element references

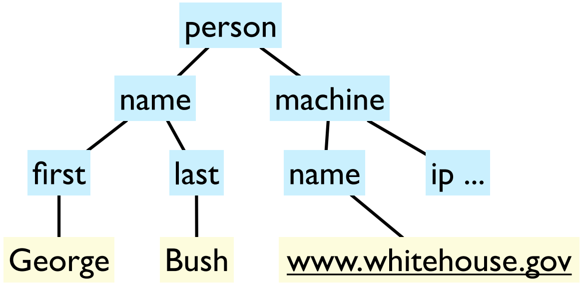
Unordered content

```
&lt;all&gt;
 &lt;element ref="A"/&gt;
 &lt;element ref="B"/&gt;
 &lt;element ref="C"/&gt;
&lt;/all&gt;
```

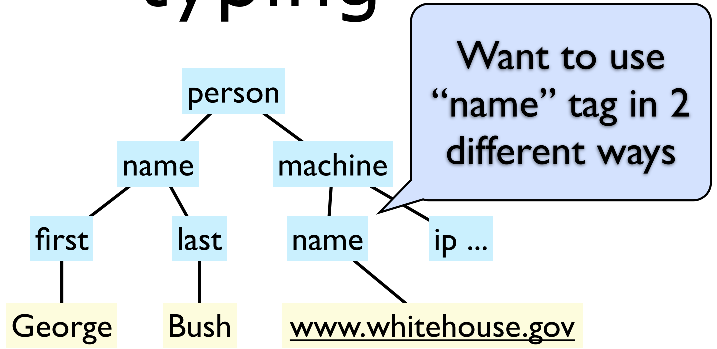


- All of A,B,C **must** appear, but can be in **any order**
- Restriction: Only distinct element references

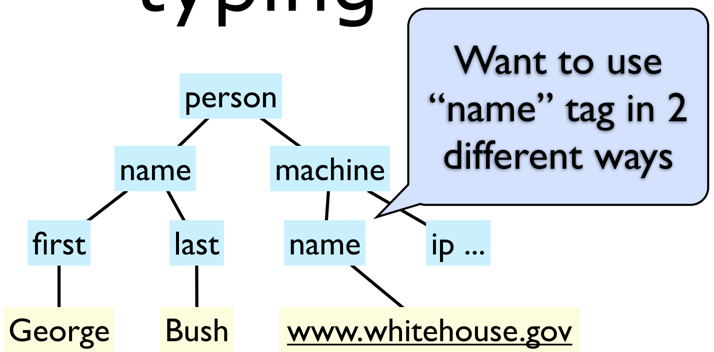
Context-sensitive typing



Context-sensitive typing



Context-sensitive typing



In DTD, would have to **badly** over-approximate:
name → (first|last|PCDATA)*

Context-sensitive typing

```
<element name="person">
 <sequence> <element name="name">
             <sequence>
              <element name="first" type="string"/>
              <element name="last" type="string"/>
             </sequence>
            </element>
            <element name="machine">
             <sequence>
              <element name="name" type="string"/>
              ... </sequence>
            </element>
 </sequence>
</element>
```

Element declarations must be consistent

- Cannot use same name in different ways within a type
- For example, this is not allowed:

```
<element name="person">
 <sequence>
  <element name="name">
   <sequence>
    <element name="first" type="string"/>
    <element name="last" type="string"/>
   </sequence>
  </element>
 <element name="name" type="string"/>
 </sequence>
</element>
```


Named groups

```
<element name="person">
 <sequence>
 <group ref="PName" />
 <element name="name" type="string"/>
 </sequence>
</element>
<group name="PName">
 <element name="name">
  <sequence>
   <element name="first" type="string"/>
   <element name="last" type="string"/>
  </sequence>
 </element>
</group>
```


Making life easier

- Extended regular expressions

  $r ::= \varnothing \mid \varepsilon \mid T \mid rs \mid r + s \mid r[n\text{-}m] \mid r \,\&\, s$

- m can be number or ∞
- T is a type name (generalizing element names)
- & means unordered concatenation (any shuffle of the two languages)
- XML schemas only allow a_1 & ... & a_n

Making life easier

- Can think of (the element part of) XML schema as a collection of type rules: T → r
  - where each type name T is associated with an element name elt(T)
  - Separating type names from element names means that elements can appear with different content in different contexts
- Cf. [Martens, Neven, Schwentick, Bex TODS 2006]

Name overloading example

Person → Name, Machine

PName → First, Last

First → string

Last → string

Machine → MName, IP

MName → string

- elt(PName) = name, elt(MName) = name, others obvious

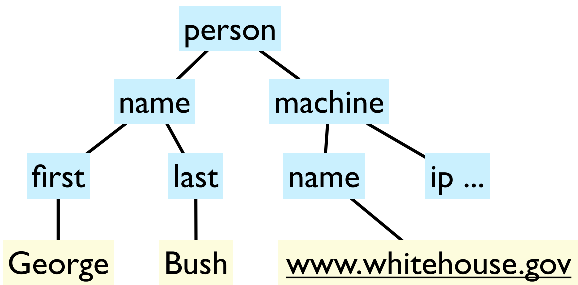
Checking validity for XML Schemas

- For DTDs, validity of each element's content can be checked independently
- For schemas, context matters
  - can be done using **tree automata**
  - efficiently, given some restrictions on rules
- all (&), cardinality constraints (slightly) complicate picture
  - naive translation to automata can be expensive
  - n! blowup for shuffle operator
  - cardinality constraints can cause exponential blowup

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

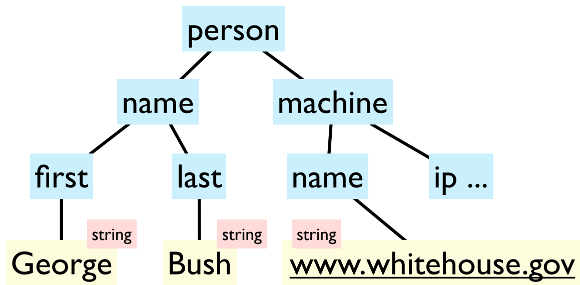


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

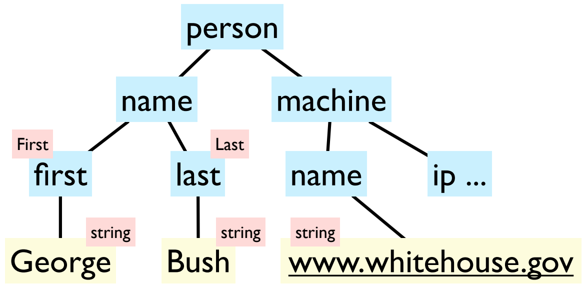


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

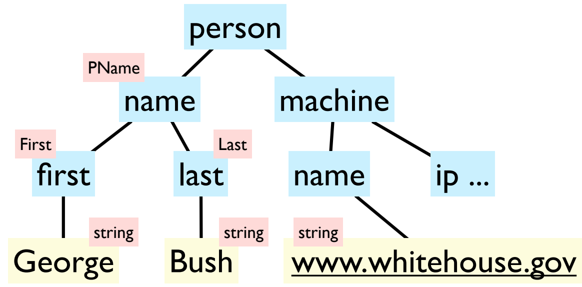


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

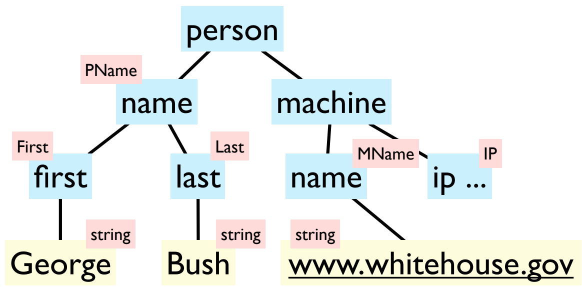


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

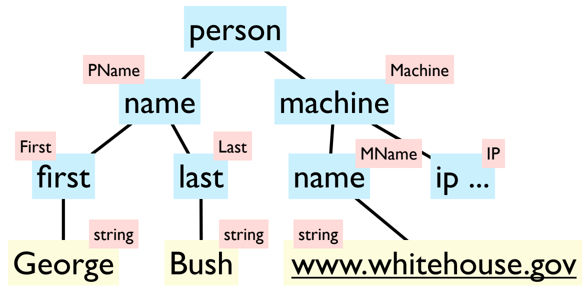


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
...

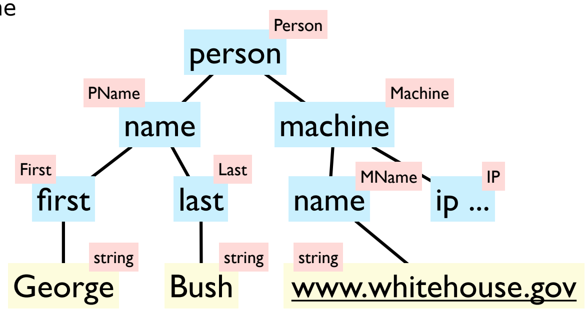


For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Example: bottom-up checking

Person → Name, Machine
PName → First, Last
First → string
Last → string
Machine → MName, IP
MName → string

elt(PName) = name,
elt(MName) = name
…



For each elt node with label L, find rule T → r such that elt(T) = L and content matches r

Keys & constraints

- Constraints are a fundamental part of the semantics of the data
  - XML may not come with a DTD/type
  - thus constraints are often the only means to specify the semantics of the data
- Constraints have proved useful in
  - semantic specifications/data modeling
  - database conversion to an XML encoding
  - data integration: information preservation
  - update anomaly prevention/consistency checking
  - normal forms for XML specifications: "BCNF", "3NF"
  - efficient storage/access, query optimization, indexing
  - …

Keys: Generalizing ID/IDREF

- XML Schema allows more general key specifications

```
<element name="people">
 <element name="person" maxOccurs="unbounded">
  <attribute name="id" type="string"/>
  <group ref="PName"/>
 </element>
</element>
<key name="person_id">
 <selector xpath="/group/person"/>
 <field xpath="@id"/>
</key>
```


Keys: Generalizing ID/IDREF

- XML Schema allows more general key specifications

```
<element name="people">
 <element name="person"
  <attribute name="id" t
  <group ref="PName"/>
 </element>
</element>
<key name="person_id">
 <selector xpath="/group/person"/>
 <field xpath="@id"/>
</key>
```

Each person element has id attribute
Must be unique **throughout document**

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

- Generalizes IDREF
  - (ID and IDREF still available as attribute types, for backward compatibility)

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

- Generalizes IDREF
  - (ID and IDREF still available as attribute types, for backward compatibility)

Each comment id must **match** one of the person ids

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

- Generalizes IDREF
  - (ID and IDREF still available as attribute types, for backward compatibility)

Local keys

- Key specifications can be made **local** to an element

```
<element name="group">
 <element name="person" maxOccurs="unbounded">
  <attribute name="id" type="string"/>
  <element ref="PName"/>
 </element>
 <key name="local_person_id">
  <selector xpath="/group/person"/>
  <field xpath="@id"/>
 </key>
</element>
```

Local keys

- Key specifications can be made **local** to an element

```
<element name="group">
 <element name="person" maxOcc
  <attribute name="id" type="s
  <element ref="PName"/>
 </element>
 <key name="local_person_id">
  <selector xpath="/group/pe
  <field xpath="@id"/>
 </key>
</element>
```

Each person element has unique id attribute **within group element**

Multiple key fields

```
<element name="people">
 <element name="person" maxOccurs="unbounded">
  <attribute name="id" type="string"/>
  <attribute name="favColor" type="string"/>
  <element ref="PName"/>
 </element>
</element>
<key name="person_id">
 <selector xpath="/group/person"/>
 <field xpath="@id"/>
 <field xpath="@favColor"/>
</key>
```


Multiple key fields

```
<element name="people">
 <element name="person" max
  <attribute name="id" type
  <attribute name="favColor
  <element ref="PName"/>
 </element>
</element>
<key name="person_id">
 <selector xpath="/group/person"/>
 <field xpath="@id"/>
 <field xpath="@favColor"/>
</key>
```

Both id and favColor must exist; no two people have the same id and same favColor

Uniqueness constraints

```
<element name="people">
 <element name="person" maxOccurs="unbounded">
  <attribute name="id" type="string"/>
  <attribute name="favColor" type="string"/>
  <element ref="PName"/>
 </element>
</element>
<unique name="person_id">
  <selector xpath="/group/person"/>
  <field xpath="@id"/>
  <field xpath="@favColor"/>
 </unique>
```

Uniqueness constraints

```
<element name="people">
 <element name="person" ma
  <attribute name="id" typ
  <attribute name="favColo
  <element ref="PName"/>
 </element>
</element>
<unique name="person_id">
  <selector xpath="/group/person"/>
  <field xpath="@id"/>
  <field xpath="@favColor"/>
 </unique>
```

One or both key fields can be missing, but must uniquely identify person if present

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

- Generalizes IDREF
  - (ID and IDREF still available as attribute types, for backward compatibility)

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

Each comment id must **match** one of the person ids

- Generalizes IDREF
  - (ID and IDREF still ava
    types, for backward co

Key references (inclusion constraints)

- Can require that all comments refer to the id of a person (according to person_id constraint)

```
<keyref name="comment" refer="person_id">
 <selector xpath=".//comment"/>
 <field xpath="@id"/>
</keyref>
```

- Generalizes IDREF
  - (ID and IDREF still available as attribute types, for backward compatibility)

Formalizing Keys for XML

- XML Schema's keys are somewhat complex
- Buneman et al. [2002, 2003] consider general forms of keys for XML, focusing on downward XPath
  - Absolute: (`/people/person`, {`@id`,`@favColor`})
  - Relative: (`/people`, (`person`, {`@id`,`@favColor`}))
  - Weak (~ `xs:unique`), strong (~ `xs:key`)
- Relative uses path to specify starting point/scope
  - whereas XML Schema keys are tied to elements/complex types
- Still an active research area, see e.g. [Hartmann & Link 2007, 2008, 2010]

Other features of XML Schema

(that we won't really use)

- Support for OO type derivation, reuse
  - Derivation by restriction
  - Groups - named parts of types
  - Import, include, redefine
- Explicit "nil" values
  - alternative to "missing"
- Default values

Limitations

- Complicated restrictions on complex types
  - regexp determinism
  - `any`
  - element description consistency
- Overall complexity daunting!
  - corollaries: limited tool support; schemas tend to be write-only
- Fortunately, most applications do not exercise all of the features

Limitations

- Complicated restrictions on complex types
  - regexp determinism
  - `any`
  - element description c
- Overall complexity d
  - corollaries: limited t write-only
- Fortunately, most applications do not exercise all of the features

What does this mean?!

If the item cannot be strictly assessed, because neither clause 1.1 nor clause 1.2 above are satisfied, [Definition:] an element information item's schema validity may be laxly assessed if its context-determined declaration is not skip by validating with respect to the ur-type definition as per Element Locally Valid (Type) (§3.3.4).

Next time

- Techniques for storing XML in relational DB
  - shredding strategies
  - query translation
- Reading: (reviews due **Monday 4pm**)
  - Relational Databases for Querying XML Documents: Limitations and Opportunities.
  - XML-SQL Query Translation Literature: The State of the Art and Open Problems
  - Accelerating XPath Evaluation in any DBMS