XML and Databases

Lecture 4  
DTDs, Schemas, Regular Expressions, Ambiguity

Sebastian Maneth  
NICTA and UNSW

CSE@UNSW -- Semester 1, 2010

### Outline

0. Comments about PRE/POST encoding  
   & about Assignment 3  (map XML to a DB)
1. DTDs
2. Regular Expressions
3. Finite-State Automata / Glushkov Automaton

### Some XPath Axes

See [http://www.w3.org/TR/xpath#axes](http://www.w3.org/TR/xpath#axes)

- **ancestor** ([n]) = { nodes on the path from root to n (wo node n)}
- **descendant** ([n]) = { nodes in the subtree rooted at n (wo node n)}
- **preceding** ([n]) = { nodes in the subtree rooted at n (wo node n)}
- **following** ([n]) = { nodes in the subtree rooted at n (wo node n)}

#### Some XPath Axes

See [http://www.w3.org/TR/xpath#axes](http://www.w3.org/TR/xpath#axes)

- ancestor(5) = \{ 1, 3 \}
- descendant(5) = \{ 6, 7 \}
- preceding(5) = \{ 2, 4 \}
- following(5) = \{ 8, 10 \}

**NOTE:** The ancestor, descendant,  
  following, preceding and self axes  
  partition a document  (ignoring attribute and namespace nodes):  
  they do not overlap and together  
  they contain all the nodes in the document.

### Some XPath Axes

See [http://www.w3.org/TR/xpath#axes](http://www.w3.org/TR/xpath#axes)

- ancestor(5) = \{ 1, 3 \}
- descendant(5) = \{ 6, 7 \}
- preceding(5) = \{ 2, 4 \}
- following(5) = \{ 8, 10 \}
Some **XPath** Axes

See [http://www.w3.org/TR/xpath#axes](http://www.w3.org/TR/xpath#axes)

- ancestor(n) = { nodes on the path from root to n (wo node n)}
- descendant(n) = { nodes in the subtree rooted at n (wo node n)}
- preceding(n) = { nodes in the subtree rooted at n (wo node n)}
- following(n) = { nodes in the subtree rooted at n (wo node n)}

ancestor(5) = { 1, 3 }  
descendant(5) = { 6, 7 }  
preceding(5) = { 2, 4 }  
following(5) = {8, 9, 10}  
self(5) = { 5 }

**Pre/Post Encoding**

<table>
<thead>
<tr>
<th>Pre</th>
<th>Post</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>c</td>
</tr>
</tbody>
</table>

### Pre/Post Encoding (Continued)

**Descendants**: Pre, Post = SELECT r1.pre FROM DOCtable r1, WHERE r1.pre < Pre AND r1.post > Post

**“structural join”**

- **findFirstChild**(pr, po) = ?
- **findFirstChild**(pr, po) = left-most node, below and to the right of (pr, po)
- or, equivalently node (pr+1, p) with p < po, if it exists.
Questions

If you know the size-of-subtree at each node, then how can you determine \( \text{nextSibling}(\text{pr}, \text{po}, \text{size}) \)?

If you know the level of each node, then how can you determine \( \text{parent}(\text{pr}, \text{po}, \text{level}) \)? And how \( \text{children}(\text{pr}, \text{po}, \text{level}) \)?

If you do not know size, but know the level of a node, then how can you determine size-of-subtree?

If you know pre/post/parent, does that also give you level and size-of-subtree?

Assignment 3

Write a program that
- reads an XML document, and a file with SQL queries
- sends a PRE/POST encoding to the DB (e.g., MySQL)
- sends the queries to the DB
- receives the answers and prints/evaluates them

- Only element/text nodes!

Nice JDBC+MySQL tutorial:
http://www.developer.com/java/data/article.php/3417381
Assignment 3

Write a program that
- reads an XML document, and a file with SQL queries
- sends a PRE/POST encoding to the DB (e.g., MySQL)
- sends the queries to the DB
- receives the answers and prints/evaluates them

Nice JDBC+MySQL tutorial:
http://www.developer.com/java/data/article.php/3417381

Assignment 3

Generate (pre,post,tag,text)-table

from the document, generate SQL insert statements

```
INSERT INTO book_tbl (pre,post,tag,text)
VALUE (1, 12, "book", null);
```

Assignment 3

Generate (pre,post,tag,text)-table & (pre.attr,value)-table

from the document, generate SQL insert statements

```
INSERT INTO book_tbl (pre,post,tag,text)
VALUE (1, 12, "book", null);
```

```xml
<pr>World</pr>
</>
```
Outline - Lectures

1. Introduction to XML, Encodings, Parsers
2. Memory Representations for XML: Space vs Access Speed
3. RDBMS Representation of XML
4. DTDs, Schemas, Regular Expressions, Ambiguity
5. Node Selecting Queries: XPath
6. Efficient XPath Evaluation
7. XPath Properties: backward axes, containment test
8. Streaming Evaluation: how much memory do you need?
9. XPath Evaluation using RDBMS
10. Properties of XPath
11. XSLT
12. XQuery
13. Wrap up, Exam Preparation etc

Outline - Assignments

2. SAX Parse into memory structure: Tree and DAG
3. Map XML into RDBMS
4. XPath evaluation
5. XPath into SQL Translation

Later in this course, we will use the PRE/POST encoding again.

We will find a systematic way to map queries on XML (XPath) into XQL queries.

Assignment 5 is about programming this mapping.
Today

XML type definition languages

want to specify a certain subset of XML doc's = a "type" of XML documents

Remember

The specification/type definition should be simple, so that

- a validator can be built automatically (and efficiently)
- the validator runs efficiently on any XML input

(similar demands as for a parser)

Type def. language must be SIMPLE!

(similarity: parsers / generators use EBNF or smaller subclasses)

O(n^3) parsing

XML Type Definition Languages

DTD (Document Type Definition, W3C)

Originated from SGML. Now part of XML

- DTD may appear at the beginning of an XML document

XML Schema (W3C)

Now at version 1.1

HUGE language, many built-in simple types

- Schemas themselves: written in XML

See the "Schema Primer" at [http://www.w3.org/TR/xmlschema-0/]

RELAX NG (Oasis)

For tree structure definition, more powerful than DTDs & Schemas

SGML relics

- only a fool does not fear "internal general parsed entities"

As an attribute value from SGML, the header of an XML document may contain a document type declaration:

```
<!DOCTYPE greeting SYSTEM "gr.dtd">
```

Or:

Store DTD in gr.dtd, and use:

```
<!DOCTYPE greeting SYSTEM "gr.dtd">
```

Example DTD

```
<!ELEMENT collection (description, recipe+)>
<!ELEMENT description (
  description, ingredient* , preparation, comment, nutrition, )>
<!ELEMENT title (ingredient, preparation, comment, nutrition, )>
<!ELEMENT ingredient (ingredient*, preparation)>
<!ELEMENT preparation (text, ingredients, preparation, comment, nutrition, )>
<!ELEMENT comment (ingredient, preparation)>
<!ELEMENT collection SYSTEM "recipes.dtd">
```

This grammar describes some obvious shortcomings:

- we cannot express that e.g. protein must contain a non-negative number
- values should only be allowed that amount is present
- the comment element should be allowed to appear anywhere
- mixed ingredient elements should only be allowed - the amount is absent

SGML relics

- only a fool does not fear "internal general parsed entities"

As an attribute value from SGML, the header of an XML document may contain a document type declaration:

```
<!DOCTYPE greeting SYSTEM "gr.dtd">
```

Or:

Store DTD in gr.dtd, and use:

```
<!DOCTYPE greeting SYSTEM "gr.dtd">
```

Example DTD

```
<!ELEMENT collection (description, recipe+)>
<!ELEMENT description (text, ingredient*, preparation, comment, nutrition, )>
<!ELEMENT title (ingredient, preparation, comment, nutrition, )>
<!ELEMENT ingredient (ingredient*, preparation)>
<!ELEMENT preparation (text, ingredients, preparation, comment, nutrition, )>
<!ELEMENT comment (ingredient, preparation)>
```

There are two kinds of recursion here...

Do you see them?
Some examples of attribute defs:

1. Fixed default attribute value

Syntax:
```xml
<!ATTLIST element-name attribute-name attribute-type #FIXED "value">
```

DTD example:
```xml
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

XML example:
```xml
<sender company="Microsoft">
```

Use if you want an attribute to have a fixed value without allowing the author to change it.

If an author includes another value, the XML parser will return an error.

Some examples of attribute defs:

2. Variable attribute value (with default)

Syntax:
```xml
<!ATTLIST element-name attribute-name attribute-type "value">
```

DTD example:
```xml
<!ATTLIST payment type CDATA "check">
```

XML example:
```xml
<payment type="check">
```

Use if you want the attribute to be present with the default value, even if the author did not include it.

Some examples of attribute defs:

2b. Enumerated attribute type

Syntax:
```xml
<!ATTLIST element-name attribute-name (value_1|value_2|...) "value">
```

DTD example:
```xml
<!ATTLIST payment type (cash|check) "cash">
```

XML examples:
```xml
<payment type="check">
```
```xml
<payment type="cash">
```

Use enumerated attribute values when you want the attribute values to be one of a fixed set of legal values.

Some examples of attribute defs:

3. Required attribute

Syntax:
```xml
<!ATTLIST element-name attribute-name attribute-type #REQUIRED>
```

DTD example:
```xml
<!ATTLIST person securityNumber CDATA #REQUIRED>
```

XML example:
```xml
<person securityNumber="3141593">
```

Use a required attribute if you don’t have an option for a default value, but still want to force the attribute to be present.

If an author forgets a required attribute, the XML parser will return an error.

Some examples of attribute defs:

4. Implied attribute

Syntax:
```xml
<!ATTLIST element-name attribute-name attribute-type #IMPLIED>
```

DTD example:
```xml
<!ATTLIST contact fax CDATA #IMPLIED>
```

XML example:
```xml
<contact fax="555-66788">
```

Use an implied attribute if you don’t want to force the author to include the attribute, and you don’t have a default value either.
The Definition of Mixed Content

- **Mixed content** is described by a repeatable OR group
  
  \[(#PCDATA | element-name | \ldots)\]^* 
  
  - Inside the group, no regular expressions – just element names
  - #PCDATA must be first, followed by 0 or more element names that are separated by |
  - The group can be repeated 0 or more times

  It should be clear how to check validity of Mixed Content!!

---

Most interesting content mode:

**Regular Expression**

An Address-Book XML Document with an Internal DTD

```xml
<addressbook>
  <person><name>John Doe</name><email>john.doe@example.com</email></person>
</addressbook>
```

The name of the DTD is `addressbook` and the syntax of the DTD is in the same file.

---

Specifying the Structure (cont’d)

1. **What is a regular expression?**
   - Given a reg. expr. how can we match a string against it?
2. **What is a finite-state automaton?**
3. **What is a deterministic regular expression?**
4. **What is a 1-unambiguous regular expression?**

The syntax of a DTD is in the same file.
Specifying the Structure (cont'd)

- So the whole structure of a person entry is specified by
  name, greet?, addr*, (tel | fax)*, email*

- This is known as a regular expression
- Why is it important?

Summary of Regular Expressions

- A The tag (i.e., element) A occurs
- e1,e2 The expression e1 followed by e2
- e* 0 or more occurrences of e
- e? Optional: 0 or 1 occurrences
- e+ 1 or more occurrences
- e | e2 either e1 or e2
- (e) grouping

Regular Expressions are a very useful concept.
- Used in EBNF, for defining the syntax of PLs
- Used in various unix tools (e.g., grep)
- Used in Perl, Tcl, text editors (like ed, emacs, ...)
- Old classical concept in CS (Stephen Kleene, 1950's)

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?
Example: 
  e = (ab | b)* a*

Finite-State Automata (FA) even more useful concept!
- They truly incarnate constant memory computation.
- Like Turing Machines, but read-only and one-way (left-to-right)
- For every Reg Exp there is a FA (and vice versa)
- Useful in many, many areas of CS (verification, compilers, learning, hardware, linguistics, UML, etc)

Finite-State Automata (FA) even more useful concept!
- They truly incarnate constant memory computation.
- Like Turing Machines, but read-only and one-way (left-to-right)
- For every Reg Exp there is a FA (and vice versa)
- For every FA there is an equivalent deterministic FA (1 per letter at most one outgoing edge)
Finite-State Automata (FA) even more useful concept!

- They truly incarnate constant memory computation.
- Like Turing Machines, but read-only and one-way (left-to-right)
- For every Reg Exp there is a FA (and vice versa)
- For every FA there is an equivalent deterministic FA

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Example:
e = (ab | b)* a*
w = a b b a a b a

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ

How can you implement a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

Construct a Finite-State Automaton

Deterministic FA: run on w takes time linear in length(w) and constant space (states, e.g., \(3 \rightarrow 1\))

Æ

Finite-State Automata (FA)

Æ

For every FA you can build an equivalent deterministic FA

- But, could become exponentially larger, if sometimes unavoidable (FA is more succinct)

Æ

For every deterministic FA you can build a minimal unique equivalent one
Thus, equivalence is decidable!
Very rare! — E.g., equivalence of EBNF’s is NOT decidable.

Æ
How can you implement a regular expression?

**Input:** Reg Expr e, string w
**Question:** Does w match e?

**Deterministic FA:** run on w takes time linear in length(w)

- Size of FA is linear in size(e) = m
- Size of DFA is exponential in m

**Algorithm**

\[ FA = BuildFA(e); \]
\[ DFA = BuildDFA(FA); \]

**Total Running time** \( O(n + 2^m) \)

- \( n = \text{length}(w) \)
- Other alternative: \( O(nm) \)

To avoid these expensive running times

W3C simply requires that \( FA = BuildFA(e) \) must be deterministic already!

Is small \( O(m) \)

- Other alternative: \( O(nm) \)

Unfortunately, we will lose some regular expressions

(which hence are not allowed to appear in a DTD!!)

How does \( BuildFA(e) \) work?

"Glushkov automaton" = "position automaton"

More details later, if time permits

Another Example

name,address*,(tel | fax)*,email*

Adding in the optional greet further complicates things

Deterministic Requirement:

Content Models must be Deterministic

- If element-type declarations are deterministic, it is easier to parse XML documents
- W3C XML recommendation requires the Glushkov automaton to be deterministic
- The states of this automaton are the positions of the regular expression (semantic actions)
- The transitions are based on the “follows set”
To summarize

In order to check whether a (large) document is valid with respect to a given DTD ("it validates") you need to:

- Check if children lists match the given Regular Expressions

This can be done efficiently, using finite-automata!

To check if a Regular Expression is allowed in a DTD we have to construct a particular finite automaton: the Glushkov automaton.

Some Things are Hard to Specify

Each employee element should contain name, age and ssn elements in some order:

```xml
<ELEMENT employee
  ( (name, age, ssn) | (age, ssn, name) )
  (ssn, name, age) ... )>
```

Suppose that there were many more fields!

Recursive DTDs

```xml
<DOCTYPE genealogy[
  <ELEMENT genealogy (person*)>
  <ELEMENT person (name, dateOfBirth, person, -- mother, -- father)
    ...
  )>
```

What is the problem with this? A parser does not notice it!

Each person should have a father and a mother. This leads to either infinite data or a person that is a descendant of herself.

Recursive DTDs

```xml
<DOCTYPE genealogy[
  <ELEMENT genealogy (person*)>
  <ELEMENT person (name, dateOfBirth, person, -- mother, -- father)
    ...
  )>
```

What is the problem with this? A parser does not notice it!
Recursive DTDs (cont’d)

The XML specification restricts regular expressions in DTDs to be deterministic (1-unambiguous).

- **Unambiguous** regular expression: "each word is witnessed by at most one sequence of positions of symbols in the expression that matches the word" [Brüggemann-Klein, Wood 1998]
  - Ambiguous expression: 
    \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
    - For \text{aaa} three witnesses: 
      \[a\text{a}\text{a}, a\text{a}\text{a}, a\text{a}\text{a}\]  
    - Unambiguous equivalent expression: 
      \[(a + b)^*\text{a}\]

- Is it enough for our purpose if the regular expression is unambiguous? No, it is not enough
  - the same unambiguous regular expression:
    \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
    - consider: \text{baa}  
      - one witness: \text{b}\text{a}\text{a} (unambiguous)  
      - it is not possible to decide \text{b}\text{a}\text{a} without looking ahead

- Without looking beyond that symbol in the input word
  \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
  - \[\text{baa}\equiv\text{b}\text{a}\text{a}\text{a}\ast\text{subscripts}\]

If a person only has a mother, how can you tell that he has a mother and does not have a father?

What is now the problem with this?

Document Type Definitions (DTDs)

- The XML specification restricts regular expressions in DTDs to be deterministic (1-unambiguous).

- Unambiguous regular expression: "each word is witnessed by at most one sequence of positions of symbols in the expression that matches the word" [Brüggemann-Klein, Wood 1998]
  - Ambiguous expression: 
    \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
    - For \text{aaa} three witnesses: 
      \[a\text{a}\text{a}, a\text{a}\text{a}, a\text{a}\text{a}\]  
    - Unambiguous equivalent expression: 
      \[(a + b)^*\text{a}\]

- Is it enough for our purpose if the regular expression is unambiguous? No, it is not enough
  - the same unambiguous regular expression:
    \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
    - consider: \text{baa}  
      - one witness: \text{b}\text{a}\text{a} (unambiguous)  
      - it is not possible to decide \text{b}\text{a}\text{a} without looking ahead

- Without looking beyond that symbol in the input word
  \[(a + b)^*\text{a}\text{a}\ast\text{subscripts}\]  
  - \[\text{baa}\equiv\text{b}\text{a}\text{a}\text{a}\ast\text{subscripts}\]

[Brüggemann-Klein, Wood 1998]:
- Can we recognize deterministic regular expressions?
  - A regular expression is deterministic (one-unambiguous) iff its Glushkov automaton is deterministic.
  - The Glushkov automaton can be computed in time quadratic in the size of the regular expression.
Glushkov’s automaton

Character in RE = state in automaton
+ one state for the beginning of the RE

Transitions show which characters/positions can precede each other

R ( E | G ) ( E X ) *

Glushkov’s automaton

- Character in RE = state in automaton
  + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

R ( E | G ) ( E X ) *

Glushkov’s automaton

- Character in RE = state in automaton
  + one state for the beginning of the RE
- Transitions show which characters/positions can precede each other

R ( E | G ) ( E X ) *
Question

Why does it take quadratic time, to construct
the Glushkov automaton for a given regular expression E?

$O(n^2)$, where n is the length of the regular expression E.

---

Question

E = $(a_1? a_2? a_3? \ldots a_n?)^*$

1) Does E contain: w = $a_1 a_3 a_2 a_1$
2) Construct the Glushkov automaton for E?
3) How many transitions (edges) does this automaton have?
4) Is there a smaller automaton which recognizes
   the same set of strings?
5) What is the smallest equivalent automaton? (merge states)

---

Question

F = $(a_1? a_2? a_3? \ldots a_n? c)^*$

1) Does F contain: v = $a_2 a_3 a$
2) How many transitions are in the Glushkov automaton for F?
3) And how many are in F's minimal automation?