XML and Databases

Lecture 4
DTDs, Schemas, Regular Expressions, Ambiguity

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

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

- the following axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes

- the preceding axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes

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

ancestor(n) = \{ nodes on the path from n to the root (wo node n) \}
descendant(n) = \{ nodes in the subtree rooted at n (wo node n) \}
preceding(n) = \{ nodes to the left of n (wo node n) and wo ancestor & descendant \}
following(n) = \{ nodes to the right of n (wo node n) and wo ancestor & descendant \}

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

See http://www.w3.org/TR/xpath#axes

ancestor(n) = { nodes on the path from n to the root (wo node n) }
descendant(n) = { nodes in the subtree rooted at n (wo node n) }
preceding(n) = { nodes to the left of n (wo node n) and wo ancestor & descendant }
following(n) = { nodes to the right of n (wo node n) and wo ancestor & descendant }

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

Pre/Post Encoding

PRE POST lab
-------------
1 7 10 a
2 1 1 b
3 7 a
4 2 2 c
5 5 d
6 3 c
7 4 c
8 6 b
9 8 b
10 9 c

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

"structural join"

firstChild(pr, po) = ?

or, equivalently
node (pr+1, po) with po < po, if it exists.
3

Questions

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

If you know the level of each node, then how can you determine parent (pr, po, level)? And how children (pr, po, 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/LEVEL 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/3427385
Assignment 3

Write a program that
→ reads an XML document, and a file with SQL queries
→ sends a PRE/POST/LEVEL 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

Assignment 3 Generate (pre,post,tag,text)-table & (pre.attr,value)-table

pre | post | level | tag  | text
--------------------------------
1   |  4   |  1   | "a"  | null
2   |  2   |  2   | "b"  | null
3   |  1   |  3   | null | "Hello World"
4   |  3   |  2   | "c"  | null

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

pre | attr | value
------------------
4 | a1 | "123"

INSERT INTO book_tbl (pre,attr,value)
VALUE (1, 12, "book", null);
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.

Outline - Assignments

2. SAX Parse into memory structure: Tree and DAG
3. Map XML into RDBMS → 27. April
4. XPath evaluation → 11. May
5. XPath into SQL Translation → 25. May

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. XSLT
11. XSLT & XQuery
12. XQuery & Updates

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. XSLT
11. XSLT & XQuery
12. XQuery & Updates
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!

(similarly: parsers generators use EBNF or smaller subclasses)

O(n^3) parsing

XML Type Definition Languages

<table>
<thead>
<tr>
<th>DTD (Document Type Definition, W3C)</th>
<th>Originated from SGML. Now part of XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTD may appear at the beginning of an XML document</td>
<td></td>
</tr>
</tbody>
</table>

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/](http://www.w3.org/TR/xmlschema-0/)

RELAX NG (Oasis)

For tree structure definition, more powerful than DTDs & Schemas

SGML relics

- only if not else all other general parsed entity

As an attribute is not parsed, the header of an XML document may contain a document type declaration:

Example DTD

```
<!ELEMENT collection (description, recipe*)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT recipe (#PCDATA)>
<!ELEMENT ingredient (ingredient, preparation, comment?, nutrition?)>
<!ELEMENT preparation (#PCDATA)>
```

There are two kinds of recursion here... Do you see them?

Example DTD

```
<!ELEMENT collection (description, recipe*)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT recipe (#PCDATA)>
```

This grammar descriptor has some obvious shortcomings:

- we cannot express that e.g. proteins must be a non-negative number
- each should only allow a certain amount in present
- the comment element should be allowed to appear anywhere
- named ingredient elements should only be allowed if amount is absent
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 example:
```xml
<payment type="check">
```

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
  
  
  \[(#PCDATA | element-name | ...)^*\]
  
  - 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!
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:
\[ e = (ab | b)* a* a \]
\[ w = a b b a a b a \]

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 vica versa)
- Useful in many, many areas of CS (verification, compilers, learning, hardware, linguistics, UML, etc, etc)

How can you implement a regular expression?
Input: Reg Expr e, string w
Question: Does w match e?
Example:
\[ e = (ab | b)* a* a \]
\[ w = a b b a a b a \]
Finite-State Automata (FA) are even more useful concepts:
- 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).

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

Finite-State Automata (FA)
For every FA you can build an equivalent deterministic FA.
But, could become exponentially larger, if sometimes unavoidable (FA is more succinct).
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.

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(e)
and constant space (states, e.g., 3 -> 1)

Finite-State Automata (FA)
For every FA you can build and equivalent deterministic FA.
But, could become exponentially larger, if sometimes unavoidable (FA is more succinct).
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.

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(e)
and constant space (states, e.g., 3 -> 1)

Algorithm
FA = BuildFA(e);
DFA = BuildDFA(FA);

Finite-State Automata (FA)
For every FA you can build and equivalent deterministic FA.
But, could become exponentially larger, if sometimes unavoidable (FA is more succinct).
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.

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(e)

Algorithm
FA = BuildFA(e);
DFA = BuildDFA(FA);

Other alternative: O(nm)

w = a b b a a b a
e = (ab | b)* a* a
Example
Input: Reg Expr e, string w
Question: Does w match e?
Deterministic FA: run on w takes time linear in length(e)
and constant space (states, e.g., 3 -> 1)
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)  
- Size of DFA is exponential in m  
- Total Running time $O(n + 2^m)$, $n = \text{length}(w)$

Other alternative: $O(nm)$

Algorithm:
1. $FA = \text{BuildFA}(e)$;  
2. $DFA = \text{BuildDFA}(FA)$;  
   
Size of $FA$ is linear in size(e) = m  
Size of DFA is exponential in m  

To avoid these expensive running times, W3C simply requires that $FA = \text{BuildFA}(e)$ must be deterministic already!

Is small! ☺ size is only $O(m)$

Unfortunately, we will lose some regular expressions (which hence are not allowed to appear in a DTD!)

Regular Expressions
- Each regular expression determines a corresponding finite-state automaton  
- Let’s start with a simpler example: name, addr*, email  

This suggests a simple parsing program

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 in the positions of the regular expression (semantic actions)
  - The transitions are based on the “follows set”
Deterministic Requirement (cont’d)

- The associated automata are succinct
- A regular language may not have an associated deterministic grammar, e.g.,
  \(<\text{ELEMENT ndeter}\)
  \(((\text{movie}|\text{director})*,\text{movie},(\text{movie}|\text{director}))>\)

This is not allowed in a DTD

To summarize

In order to check whether a (large) document is valid with a given DTD ("it validates") you need to
→ Check if children lists match the given Reg Expr’s

This can be done efficiently, using finite-automata!

Some Things are Hard to Specify

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

\(<\text{ELEMENT employee}\>
  \(((\text{name, age, ssn})|((\text{age, ssn, name})|\>
  \(((\text{ssn, name, age})|\>
\>)

Suppose that there were many more fields!

Recursive DTDs

\(<\text{DOCTYPE genealogy}\>
  \(<\text{ELEMENT genealogy (person)*}\>\>
  \(<\text{ELEMENT person}\>
      \text{name, dateOfBirth, person, -- mother}
      \text{person, -- father}
  \>)

\(\)

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

Recursive DTDs

\(<\text{DOCTYPE genealogy}\>
  \(<\text{ELEMENT genealogy (person)*}\>\>
  \(<\text{ELEMENT person}\>
      \text{name, dateOfBirth, person, -- mother}
      \text{person, -- 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.
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 \((a + b)^*a\) has witness \(a^*b\) and \(a^*b^*a\)
- For \(aaa\) three witnesses: \(a^*a\), \(a^*a^*\), \(a^*a^*a\)
- Unambiguous equivalent expression: \((a + b)^*a\)

Similarly, \((a + b)^*a^+\) is not unambiguous, but \((a + b)^*a^*\) is.

Consider: \(baa\)
- One witness: \(b + a + a\) (unambiguous)
- It is not possible to decide \(b + a^+\) without looking ahead

Without looking beyond that symbol in the input word

\((a + b)^*a\) unambiguous 1-unambiguous

\((a + b)^*a^*\) unambiguous 1-unambiguous

Can you find a 1-unambiguous Reg Exp for \((a + b)^*a^*\) not so easy...
Glushkov’s automaton

\[ R(\text{E}|\text{G})(\text{EX})^* \]

Character in RE = state in automaton

\[ R(\text{E}|\text{G})(\text{EX})^* \]

Glushkov’s automaton

Character in RE = state in automaton

+ one state for the beginning of the RE

\[ R(\text{E}|\text{G})(\text{EX})^* \]

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

\[ R(\text{E}|\text{G})(\text{EX})^* \]

\[ R(\text{E}|\text{G})(\text{EX})^* \]

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

\[ R(\text{E}|\text{G})(\text{EX})^* \]

\[ R(\text{E}|\text{G})(\text{EX})^* \]
Questions

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? ($\rightarrow$ merge states)

---

**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 $E$.

---

**END Lecture 4**