Applied Databases

Lecture 3
DTDs (regular expressions) & DOM

Sebastian Maneth

University of Edinburgh - January 23rd, 2017
Outline

1. **DTD**  Regular Expression → Glushkov Automaton
2. **DOM**  Document Object Model
1. Regular Expressions

`<!DOCTYPE addressbook [
  <!ELEMENT addressbook (person*) >
  <!ELEMENT person (name,greet*,address*,(fax|tel)*,email*)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT greet (#PCDATA)>
  <!ELEMENT address (#PCDATA)>
  <!ELEMENT fax (#PCDATA)>
  <!ELEMENT tel (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
]>`
1. Regular Expressions

- choice: ( . . | . . | . . )

- sequence: ( . . , . . , . . )

- optional: . . . ?

- zero or more: . . . *

- one or more: . . . +

- element names

Note

→ #PCDATA may not appear in these regular expressions!

→ use mixed content instead
Regular Expressions are a very useful concept
→ used in EBNF, for defining the syntax of PLs
→ used in various unix tools (e.g., grep)
→ supported in most PLs (esp. Perl), text editors
→ classical concept in CS (Stephen Kleene, 1950’s)

How can you **implement** a regular expression?

Input: RegEx e, string w
Question: Does w match e?  
→ use Finite Automata (FA)

Example
\[ e = (ab | b)^* a^* a \]
\[ w = a b b a a b a \]

| match? |
Finite-State Automata (FA)

→ *constant memory computation*

→ as Turing Machines, but *read-only* and *one-way* (left-to-right)

for every ReEx there is a FA (and vice versa)

Deterministic FA (DFA) = no two outgoing edges with same label

DFA Matching: time O( |DFA| + |w| )
“one finger needed”

FA Matching: time O(|FA| + m * |w| )
“at most #states many fingers needed”
→ every FA can be effectively transformed into an equivalent DFA.

→ can take exponential time! ("subset construction")

How can you implement a regular expression?

Input: RegEx e, string w
Question: Does w match e?

FA = BuildFA(e);
FA.run;
→ or
DFA = BuildDFA(FA);
DFA.run

Running time  O(m + mn)
or  O(m + 2^m + n)
Regular expressions $e$ for which $\text{BuildFA}(e)$ is deterministic are called \textit{deterministic regular expressions}.

$\quad \rightarrow$ max number of transitions (edges) for \textit{m states} and \textit{k symbols}? 

builds Glushkov automaton
To avoid these expensive running times

W3C requires that $\text{BuildFA}(e)$ must be deterministic!

Is small! 😊

size is only $O(m^2)$

**Running time** $O(km + n)$

W3C DTD-definition

builds Glushkov automaton

Regular expressions $e$ for which $\text{BuildFA}(e)$ is deterministic are called **deterministic regular expressions**.

max number of transitions (edges) for $m$ states and $k$ symbols?
\[
\text{BuildFA}(e) = \text{"Glushkov Automaton"} = \text{"Position Automaton"} \quad [\text{Glushkov1961}]
\]

\[
\text{BuildFA}(e) = \text{every letter in } e \text{ becomes a state}
\]

\[
(a\mid b)^*aa^*
\]

\[
0 \quad 1 \quad 2 \quad 3 \quad 4
\]

\[
\text{plus extra start-state } \text{"0"}
\]
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ identify end-position(s)

BuildFA(e) = every letter in e becomes a state
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ which positions are reachable from “position 0”?

BuildFA(e) = every letter in e becomes a state
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ which positions are reachable from positions 1 and 2?

BuildFA(e) = every letter in e becomes a state

(a | b)*aa*
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ which positions are reachable from position 3?

BuildFA(e) = every letter in e becomes a state
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ which positions are reachable from position 4?

→ Done!

BuildFA(e) = every letter in e becomes a state

(a | b)*aa*
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ a “successful run” for the input word “aaa”

BuildFA(e) = every letter in e becomes a state

(a | b)*aa*
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ a “successful run” for the input word “aaa”

→ how many other successful runs are there for “aaa”?

BuildFA(e) = every letter in e becomes a state

(a | b)*aa*
BuildFA(e) = “Glushkov Automaton” = “Position Automaton” [Glushkov1961]

→ a “successful run” for the input word “aaa”

→ how many other successful runs are there for “aaa”?

BuildFA(e) = every letter in e becomes a state

\[(a \mid b)^*aa^*\]
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: 
  
  $$(a | b)^*aa^* \xrightarrow{\text{mark with subscripts}} (a_1 | b_1)^*a_2a_3^*$$

- For $aaa$, three witnesses: $a_1a_1a_2$, $a_1a_2a_3$, $a_2a_3a_3$

- Unambiguous equivalent expression: $(a | b)^*a$
Document Type Definitions (DTDs)

• The XML specification restricts regular expressions in DTDs to be deterministic \textit{(1-unambiguous)}.

• \textbf{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 \((a | b)^*aa^*\) \(\xrightarrow{\text{mark with subscripts}}\) \((a_1 | b_1)^*a_2a_3^*\)

  ✓ For \textbf{aaa}

  \(\xrightarrow{\text{three witnesses:}}\)

  \(a_1a_1a_2, a_1a_2a_3, a_2a_3a_3\)

  ✓ Unambiguous equivalent expression \(\textbf{(a | b)*a}\) \(\not\text{1-unambiguous!}\)

→ \textbf{1-unambiguous}: decide \textbf{position} by looking \textbf{only at current symbol}
consider \textbf{baa}: \textbf{b}_{1}a?
Questions for each expression, deterministic or not?

→ a?b?

→ a?b?a

→ a(aba)*b

→ (a?b?c?d?e?)*
Questions for each expression, deterministic or not?

→ a?b?

→ a?b?a

→ a(aba)*b

→ (a?b?c?d?e?)*

How many edges in the Glushkov automaton? (a1?a2? .... ak?) for distinct a1,a2,...
Questions for each expression, deterministic or not?

→ a?b?

→ a?b?a

→ a(aba)*b

→ (a?b?c?d?e?)*


How many edges in the Glushkov automaton?

(a1?a2? … ak?) for distinct a1,a2,…

→ (a | b)*a is not deterministic.

Can you find an equivalent expression that is deterministic?
Questions for each expression, deterministic or not?

→ a?b?

→ a?b?a

→ a(aba)*b

→ (a?b?c?d?e?)* How many edges in the Glushkov automaton? (a1?a2? …. ak?) for distinct a1,a2,...

→ (a | b)*a is not deterministic.

Can you find an equivalent expression that is deterministic?

→ (a | b)*a(a | b) is not deterministic.

Can you find an equivalent expression that is deterministic?
Questions  for each expression, deterministic or not?

→  a?b?

→  a?b?a

→  a(aba)*b

→  (a?b?c?d?e?)*

How many edges in the Glushkov automaton? (a1?a2? …. ak?) for distinct a1,a2,...

Notes

→  there exist regular expressions for which no equivalent deterministic expression exists

→  this can be decided, and an equivalent deterministic reg expr constructed if it exists  [Brüggemann-Klein, Wood 1998]
XML Parsers

- Document Object Model - DOM
- Simple API for XML - SAX
XML Parsers

- Document
  - DOM parser
  - SAX parser
  - Stream of events

- Object API
  - Complex applications, XSLT, editor
  - Simple applications, very efficient
XML Parsers

→ DOM - loads *full document* into memory

→ SAX - generates streaming events
  - by default: *nothing* stored in memory
2. DOM – Document Object Model

→ Language and platform-independent view of XML

→ DOM APIs exist for many PLs (Java, C++, C, Python, JavaScript …)

DOM relies on two main concepts

(1) The XML processor constructs the complete XML document tree (in-memory)

(2) The XML application issues DOM library calls to explore and manipulate the XML tree, or to generate new XML trees.

Advantages

• easy to use
• once in memory, no tricky issues with XML syntax anymore
• all DOM trees serialize to well-formed XML (even after arbitrary updates)!
2. DOM – Document Object Model

→ Language and platform-independent view of XML

→ DOM APIs exist for many PLs (Java, C++, C, Python, JavaScript …)

DOM relies on two main concepts

(1) The XML processor constructs the **complete XML document tree** (in-memory)

(2) The XML application issues DOM library calls to **explore** and **manipulate** the XML tree, or to **generate** new XML trees.

**Advantages**

• easy to use

• once in memory, no tricky issues with XML syntax anymore

• all DOM trees serialize to well-formed XML (even after arbitrary updates)!

**Disadvantage**

Uses LOTS of memory!!
2. DOM – Document Object Model

A NamedNodeMap

“ItemID” → a1
“xyz” → a2

This is also a text node t1 which is the firstChild of a1

```
31

2. DOM – Document Object Model

Items

firstChild
parentNode
nextSibling
previousSibling

n1

Item

n2

Item

n3

Item

n4

Item

n5

Name

“screwdriver”

attributes

ItemID nodeValue

“107378154”

A NamedNodeMap

“ItemID” → a1
“xyz” → a2

This is also a text node t1 which is the firstChild of a1
```
2. DOM – Document Object Model

- Item
  - Item
  - Item
  - Item
  - Item

- Items
  - firstChild
  - parentNode
  - nextSibling
  - previousSibling
  - lastChild

- attributes

- "ItemID" → a1
  - "xyz" → a2

- A NamedNodeMap

- ItemID
  - nodeValue → "107378154"

→ how much memory is needed for a typical XML document of 1GB?
2. DOM – Document Object Model

Example

<table>
<thead>
<tr>
<th>XML size</th>
<th>DOM process size</th>
</tr>
</thead>
<tbody>
<tr>
<td>81M</td>
<td>$81M \times 2 \rightarrow 164M$</td>
</tr>
<tr>
<td></td>
<td>Text only, with one embracing element</td>
</tr>
<tr>
<td>52M</td>
<td>$52M \times 13 \rightarrow 680M$</td>
</tr>
<tr>
<td></td>
<td>Treebank, deep tree structure, short tagnames</td>
</tr>
</tbody>
</table>

→ how much memory is needed for a typical XML document of 1GB?
### DOM Level 1 (Core)

#### Some methods

<table>
<thead>
<tr>
<th>DOM type</th>
<th>Method</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong></td>
<td><strong>nodeName</strong></td>
<td>redefined in subclasses</td>
</tr>
<tr>
<td></td>
<td><strong>nodeValue</strong></td>
<td>: DOMString</td>
</tr>
<tr>
<td></td>
<td><strong>parentNode</strong></td>
<td>: Node</td>
</tr>
<tr>
<td></td>
<td><strong>firstChild</strong></td>
<td>: Node</td>
</tr>
<tr>
<td></td>
<td><strong>lastChild</strong></td>
<td>: Node</td>
</tr>
<tr>
<td></td>
<td><strong>nextSibling</strong></td>
<td>: Node</td>
</tr>
<tr>
<td></td>
<td><strong>previousSibling</strong></td>
<td>: Node</td>
</tr>
<tr>
<td></td>
<td><strong>childNodes</strong></td>
<td>: NodeList</td>
</tr>
<tr>
<td></td>
<td><strong>attributes</strong></td>
<td>: NamedNodeMap</td>
</tr>
<tr>
<td></td>
<td><strong>ownerDocument</strong></td>
<td>: Document</td>
</tr>
<tr>
<td></td>
<td><strong>replaceChild</strong></td>
<td>: Node</td>
</tr>
<tr>
<td><strong>Document</strong></td>
<td><strong>createElement</strong></td>
<td>: Element</td>
</tr>
<tr>
<td></td>
<td><strong>createComment</strong></td>
<td>: Comment</td>
</tr>
<tr>
<td></td>
<td><strong>getElementsByTagName</strong></td>
<td>: NodeList</td>
</tr>
</tbody>
</table>
Character strings (DOM type DOMString) are defined to be encoded using UTF-16 (e.g., Java DOM represents type DOMString using its String type).
DOM Level 1 (Core)

*Name, Value, and attributes* depend on the *type* of the current node.

The values of `nodeName`, `nodeValue`, and `attributes` vary according to the node type as follows:

<table>
<thead>
<tr>
<th>Node Type</th>
<th><code>nodeName</code></th>
<th><code>nodeValue</code></th>
<th><code>attributes</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td><code>tagName</code></td>
<td><code>null</code></td>
<td><code>NamedNodeMap</code></td>
</tr>
<tr>
<td>Attr</td>
<td><code>name of attribute</code></td>
<td><code>value of attribute</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>Text</td>
<td><code>#text</code></td>
<td><code>content of the text node</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>CDATASection</td>
<td><code>#cdata-section</code></td>
<td><code>content of the CDATA Section</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>EntityReference</td>
<td><code>name of entity referenced</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>Entity</td>
<td><code>entity name</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td><code>target</code></td>
<td><code>entire content excluding the target</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>Comment</td>
<td><code>#comment</code></td>
<td><code>content of the comment</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>Document</td>
<td><code>#document</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>DocumentType</td>
<td><code>document type name</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>DocumentFragment</td>
<td><code>#document-fragment</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
<tr>
<td>Notation</td>
<td><code>notation name</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
</tr>
</tbody>
</table>
DOM Level 1 (Core)

Some details

Creating an *element/attribute* using `createElement/createAttribute` does not wire the new node with the XML tree structure yet.

→ Call `insertBefore, replaceChild, …,` to wire a node at an explicity position.

DOM type *NodeList* makes up for the lack of collection data types in many programming languages.

DOM type *NamedNodeMap* represents an *association table* (nodes may be accessed by name).

Example:

```
getterbutes
v0  a1
firstChild
lastChild

“17”

A NamedNodeMap
```

NodeValue

Methods: `getNamedItem, setNamedItem`, …

```
“ItemID”  → a1
“xyz”     → a2
```

text node! (with `nodeValue=“17”`)

public static void recursiveDescent(Node n, int level) {

    // adjust indentation according to level
    for(int i=0; i<4*level; i++) System.out.print(" ");

    // dump out node name, type, and value
    String ntype = typeName[n.getNodeType()];
    String nname = n.getNodeName();
    String nvalue = n.getNodeValue();
    System.out.println("Type = " + ntype + ", Name = " + nname + ", Value = " + nvalue);

    // dump out attributes if any
    org.w3c.dom.NamedNodeMap nattrib = n.getAttributes();
    if(nattrib != null && nattrib.getLength() > 0)
        for(int i=0; i<nattrib.getLength(); i++)
            recursiveDescent(nattrib.item(i), level+1);

    // now walk through children list
    org.w3c.dom.NodeList nlist = n.getChildNodes();
    for(int i=0; i<nlist.getLength(); i++)
        recursiveDescent(nlist.item(i), level+1);
}
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello"> 
  <!ENTITY hi1 "&hi;&hi;"> 
  <!ENTITY hi2 "&hi1;&hi1;"> 
  <!ENTITY hi3 "&hi2;&hi2;"> 
  <!ENTITY s "<d></d>" ]>
<a a1='17' a2='29'><b>xy &hi3; world &s; zz</b></a>

$ java MyDOM file.xml
Successfully parsed - file.xml
Type = Document, Name = #document, Value = null
  Type = DocType, Name = greeting, Value = null
  Type = Element, Name = a, Value = null
    Type = Attr, Name = a1, Value = 17
      Type = Text, Name = #text, Value = 17
    Type = Attr, Name = a2, Value = 29
      Type = Text, Name = #text, Value = 29
  Type = Element, Name = b, Value = null
    Type = Text, Name = #text, Value = xy HelloHelloHelloHello world
  Type = Element, Name = d, Value = null
    Type = Text, Name = #text, Value = zz
END

Lecture 3