Applied Databases

Lecture 2
Document Type Definitions (DTDs)

Sebastian Maneth

University of Edinburgh - January 14th, 2016

Outline

1. DTD Document Type Definition

Recap

XML → widely adopted data exchange format

→ lingua franca for data on the web

→ ordinary text files

→ describe tree structures (even DAGs, if ID/IDREF used)

→ `<tag>` … `</tag>` describes an element node labeled “`tag`”

Well-Formed XML

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity *references* (two kinds)

character reference
Type <key>less-than</key>
(<) to save options.

<family rel="brother",age="25">
<name>
…
</family>

This document was prepared on &docdate; and

→ document must have a root-node (aka "document node")
i.e., document must start with "<"-character!

Well-Formed XML

Assignment 1
only element nodes and attributes

→ attributes
→ processing instructions
→ comments `<!-- some comment -->`
→ namespaces
→ entity *references* (two kinds)

character reference
Type <key>less-than</key>
(<) to save options.

<family rel="brother",age="25">
<name>
…
</family>

This document was prepared on &docdate; and

→ document must have a root-node (aka "document node")
i.e., document must start with "<"-character!

Well-Formed XML

Well-formed or not? Context-free or context-sensitive violation?

→ <a><b/

→ <a> is a shorthand for

→ <a>

→ <<a>><>

→

→ <a><a/>

→ <a>

XML Grammar - EBNF-style

```
[1]    document    ::=   prolog element Misc*
[2]        Char    ::=   a Unicode character
[3]           S    ::=   (‘ ’ | ‘\t’ | ‘\n’ | ‘\r’)+
[4]    NameChar    ::=   (Letter | Digit | ‘.’ | ‘-’ | ‘:’
[5]        Name    ::=   (Letter | '_' | ':') (NameChar)*

[22]     prolog    ::=   XMLDecl? Misc* (doctypedecl Misc*)?
[23]    XMLDecl    ::=   '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>‘
[24]VersionInfo    ::=   S'version'Eq("'"VersionNum"'"|'"'VersionNum'"')
[25]         Eq    ::=   S? '=' S?
[26]VersionNum     ::=   '1.0‘

[39]    element    ::=   EmptyElemTag
                       | STag content Etag
[40]       STag    ::=   '<' Name (S Attribute)* S? '>'
[41]  Attribute    ::=   Name Eq AttValue
[42]       ETag    ::=   '</' Name S? '>‘
[43]    content    ::=   (element | Reference | CharData?)*
[44]EmptyElemTag   ::=   '<' Name (S Attribute)* S? '/>‘

[67] Reference     ::=   EntityRef | CharRef
[68] EntityRef     ::=   '&' Name ';‘
[84] Letter        ::=   [a-zA-Z]
[88] Digit         ::=   [0-9]
```

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*)

(similarly: parser 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
→ or be included from a file: `<!DOCTYPE xyz SYSTEM "xyz.dtd">`

XML Schema ("XSD") (W3C)
Now at version 1.1
HUGE language, many built-in simple types

→ Schemas: written in XML

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

RELAX NG (Oasis)
Wrt tree structure definition, more powerful than DTDs & Schemas

SGML relics

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

As an unfortunate heritage from SGML, the header of an XML document may contain a **document type declaration**:

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
  <!ATTLIST greeting style (big|small) "small">
  <!ENTITY hi "Hello">
]>
<greeting> &hi; world! </greeting>
```

This part can contain:

- DTD (Document Type Definition) information:
  - element type declarations (**ELEMENT**)
  - attribute-list declarations (**ATTLIST**)

  (described later...)
- entity declarations (**ENTITY**) - a simple macro mechanism
- notation declarations (**NOTATION**) - data format specifications

Avoid all these features whenever possible!

Unfortunately, they cannot always be ignored - all XML processors (even non-validating ones) are required to:

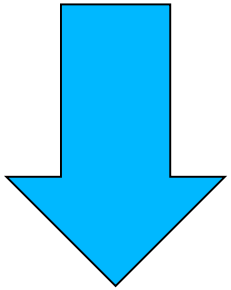
- normalize attribute values (prune white-space etc.) ← if the attribute type is not CDATA
- handle internal entity references (e.g. expand **&hi;** in **greeting**)
- insert default attribute values (e.g. insert **style="small"** in **greeting**)

according to the document type declaration, if a such is present.

(also next 4 slides)

DTDs

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
  <!ATTLIST greeting style ( big | small ) "small">
  <!ENTITY hi "Hello">
]>
<greeting> &hi; World! </greeting>
```



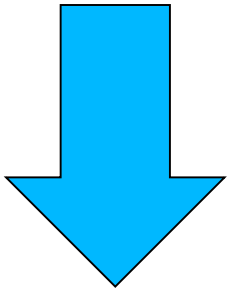
XML Processor

```
<?xml version="1.0"?>
<greeting style="small"> Hello World! </greeting>
```

inserted by Processor

DTDs

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY footer SYSTEM "/boilerplate/footer.xml">
]>
<greeting> &hi; World! </greeting>
```

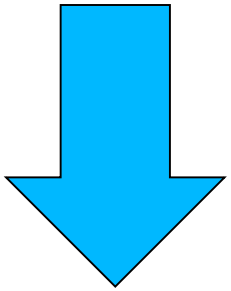


XML Processor

```
<?xml version="1.0"?>
<greeting style="small"> <footer><picture loc="xyz"></picture> ..
                          </footer> World! </greeting>
```

DTDs

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY footer SYSTEM "/boilerplate/footer.xml">
]>
<greeting> &footer; world! </greeting>
```



XML Processor

```
<?xml version="1.0"?>
<greeting style="small"> <footer><picture loc="xyz"></picture> ..
                          </footer> world! </greeting>
```

- → `footer.xml` is an *external general parsed entity*
- → `footer.xml` is NOT a well-formed XML document, e.g., it need not have a root element!
- → Processor may (or not) replace `&hi;` by `footer.xml`

DTDs

External general parsed entities

→ must be well-formed,
if an extra element node was wrapped around it

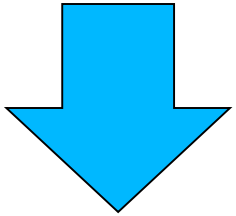
→ at the limits of what you can comfortably fit in a DTD

→ web sites prefer to store repeated content in external files and load it into their pages using PHP, server-side includes, or some similar.

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
]>
<greeting> &hi1; World! </greeting>
```

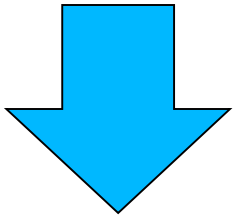


```
<?xml version="1.0"?>
<greeting> HelloHello World! </greeting>
```

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
  <!ENTITY hi2 "&hi1;&hi1;">
  <!ENTITY hi3 "&hi2;&hi2;">
]>
<greeting> &hi3; World! </greeting>
```



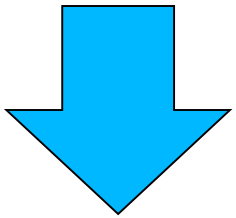
```
<?xml version="1.0"?>
<greeting> HelloHelloHelloHelloHelloHelloHelloHello World! </greeting>
```

8 times

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
  <!ENTITY hi2 "&hi1;&hi1;">
  <!ENTITY hi3 "&hi2;&hi2;">
  ...
  <!ENTITY hi10 "&hi2;&hi2;">
]>
<greeting> &hi10; World! </greeting>
```



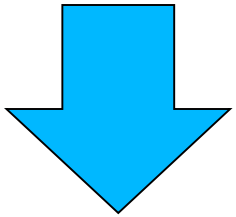
```
<?xml version="1.0"?>
<greeting> HelloHello . . . . . HelloHelloHelloHello World! </greeting>
```

1024 times

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
...
  <!ENTITY hi10 "&hi2;&hi2;">
]>
<greeting> &hi10; World! </greeting>
```



exponential size increase

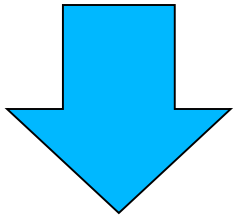
```
<?xml version="1.0"?>
<greeting> Hello. . . Hello World! </greeting>
```

→ Validation / parsing may take exponential wrt size of the input (but linear wrt size of output). ☺

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi1;&hi;">
]>
<greeting> &hi1; World! </greeting>
```

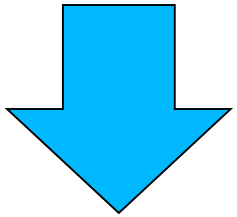


?

DTDs

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi1;&hi;">
]>
<greeting> &hi1; World! </greeting>
```



Processor reports error
[circular ENTITY declaration]

DTDs

Entity Expansion

→ can take exponential time

→ need to check for circularities (non-termination)

- `<!DOCTYPE root-element [ doctype-declaration... ]>`
  determines the name of the root element and contains the document type declarations

- `<!ELEMENT element-name content-model>`
  associates a *content model* to all elements of the given name

  content models:

  - `EMPTY`: no content is allowed
  - `ANY`: any content is allowed
  - `(#PCDATA|element-name|...)*`: "mixed content", arbitrary sequence of character data and listed elements
  - *deterministic regular expression over element names*: sequence of elements matching the expression
    - choice: `(...|...|...)`
    - sequence: `(...,...,...)`
    - optional: `...?`
    - zero or more: `...*`
    - one or more: `...+`

- `<!ATTLIST element-name attr-name attr-type attr-default ...>`
  declares which attributes are allowed or required in which elements

  attribute types:

  - `CDATA`: any value is allowed (the default)
  - `(value|...)`: enumeration of allowed values
  - `ID`, `IDREF`, `IDREFS`: ID attribute values must be unique (contain "element identity"), IDREF attribute values must match some ID (reference to an element)
  - `ENTITY`, `ENTITIES`, `NMTOKEN`, `NMTOKENS`, `NOTATION`: just forget these... (consider them deprecated)

  attribute defaults:

  - `#REQUIRED`: the attribute must be explicitly provided
  - `#IMPLIED`: attribute is optional, no default provided
  - `"value"`: if not explicitly provided, this value inserted by default
  - `#FIXED "value"`: as above, but only this value is allowed

This is a simple subset of SGML DTD.

Validity can be checked by a simple top-down traversal of the XML document (followed by a check of IDREF requirements).

Some examples of attribute defs:

(1) Fixed default attribute value

Syntax:

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

DTD example:

```
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

XML example:

```
<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:

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

DTD example:

```
<!ATTLIST payment type CDATA "check">
```

XML example:

```
<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:

```
<!ATTLIST element-name attribute-name (value1|value2|..) “value”>
```

DTD example:

```
<!ATTLIST payment type (cash|check) "cash">
```

XML example:

```
<payment type="check">
or <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:

```
<!ATTLIST element-name attribute_name attribute-type #REQUIRED>
```

DTD example:

```
<!ATTLIST person securityNumber CDATA #REQUIRED>
```

XML example:

```
<person securityNumber="3141593">
```

must be included

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:

```
<!ATTLIST element-name attribute_name attribute-type #IMPLIED>
```

DTD example:

```
<!ATTLIST contact fax CDATA #IMPLIED>
```

XML example:

```
<contact fax="555-667788">
```

may be included

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.

- `<!DOCTYPE root-element [ doctype-declaration... ]>`
  determines the name of the root element and contains the document type declarations

- `<!ELEMENT element-name content-model>`
  associates a *content model* to all elements of the given name

  content models:

  - `EMPTY`: no content is allowed
  - `ANY`: any content is allowed
  - `(#PCDATA|element-name|...)*`: "mixed content", arbitrary sequence of character data and listed elements
  - *deterministic regular expression over element names*: sequence of elements matching the expression
    - choice: `(...|...|...)`
    - sequence: `(...,...,...)`
    - optional: `...?`
    - zero or more: `...*`
    - one or more: `...+`

- `<!ATTLIST element-name attr-name attr-type attr-default ...>`
  declares which attributes are allowed or required in which elements

  attribute types:

  - `CDATA`: any value is allowed (the default)
  - `(value|...)`: enumeration of allowed values
  - `ID`, `IDREF`, `IDREFS`: ID attribute values must be unique (contain "element identity"), IDREF attribute values must match some ID (reference to an element)
  - `ENTITY`, `ENTITIES`, `NMTOKEN`, `NMTOKENS`, `NOTATION`: just forget these... (consider them deprecated)

  attribute defaults:

  - `#REQUIRED`: the attribute must be explicitly provided
  - `#IMPLIED`: attribute is optional, no default provided
  - `"value"`: if not explicitly provided, this value inserted by default
  - `#FIXED "value"`: as above, but only this value is allowed

This is a simple subset of SGML DTD.

Validity can be checked by a simple top-down traversal of the XML document (followed by a check of IDREF requirements).

- `<!DOCTYPE root-element [ doctype-declaration... ]>`
  determines the name of the root element and contains the document type declarations

- `<!ELEMENT element-name content-model>`
  associates a *content model* to all elements of the given name

  content models:

  - **`EMPTY`**: no content is allowed
  - **`ANY`**: any content is allowed
  - `(#PCDATA|element-name|...)*`: "mixed content", arbitrary sequence of character data and listed elements
  - *deterministic regular expression over element names*: sequence of elements matching the expression
    - choice: `(...|...|...)`
    - sequence: `(...,...,...)`
    - optional: `...?`
    - zero or more: `...*`
    - one or more: `...+`

- `<!ATTLIST element-name attr-name attr-type attr-default ...>`
  declares which attributes are allowed or required in which elements

  attribute types:

  - **`CDATA`**: any value is allowed (the default)
  - `(value|...)`: enumeration of allowed values
  - **`ID`**, **`IDREF`**, **`IDREFS`**: ID attribute values must be unique (contain "element identity"), IDREF attribute values must match some ID (reference to an element)
  - **`ENTITY`**, **`ENTITIES`**, **`NMTOKEN`**, **`NMTOKENS`**, **`NOTATION`**: just forget these... (consider them deprecated)

  attribute defaults:

  - **`#REQUIRED`**: the attribute must be explicitly provided
  - **`#IMPLIED`**: attribute is optional, no default provided
  - `"value"`: if not explicitly provided, this value inserted by default
  - **`#FIXED`** `"value"`: as above, but only this value is allowed

How??

This is a simple subset of SGML DTD.

Validity can be checked by a simple top-down traversal of the XML document (followed by a check of IDREF requirements).

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)>
]>
```

Regular Expressions

- choice: (.. | .. | ..)
- sequence: (.., .., ..)
- optional: ...?
- zero or more: ...*
- one or more: ...+
- element names

Note

→ #PCDATA may **not** appear in these regular expressions!

Regular Expressions

- zero or more: **...*

```
<!DOCTYPE a [
  <!ELEMENT a (b*)>
  <!ELEMENT b (#PCDATA)>
]>
<a><b>abcdefg</b></a>
```

valid document

Regular Expressions

- zero or more: `...*`

```
<!DOCTYPE a [
  <!ELEMENT a (b*)>
  <!ELEMENT b (#PCDATA)>
]>
<a><b></b></a>
```

valid document

careful: there is NO empty text node here

Regular Expressions

- zero or more: `...*`

```
<!DOCTYPE a [
  <!ELEMENT a (b*)>
  <!ELEMENT b (#PCDATA)>
]>
<a></a>
```

→ is it valid??

Regular Expressions

- zero or more: **...*****

```
<!DOCTYPE a [
  <!ELEMENT a (b*)>
  <!ELEMENT b (#PCDATA)>
]>
<a>abcde</a>
```


not valid

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!

Example DTD

A DTD for our recipe collections, `recipes.dtd`:

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

<!ELEMENT description ANY>

<!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED
                     unit CDATA #IMPLIED>

<!ELEMENT preparation (step*)>

<!ELEMENT step (#PCDATA)>

<!ELEMENT comment (#PCDATA)>

<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition protein CDATA #REQUIRED
                    carbohydrates CDATA #REQUIRED
                    fat CDATA #REQUIRED
                    calories CDATA #REQUIRED
                    alcohol CDATA #IMPLIED>
```

There are **two** kinds of *recursion* here..

Do you see them?

By inserting:

```
<!DOCTYPE collection SYSTEM "recipes.dtd">
```

in the headers of recipe collection documents, we state that they are intended to conform to `recipes.dtd`.

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)

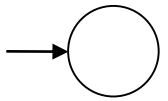
How can you **implement** a regular expression?

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

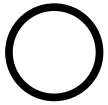
Example
e = (ab | b)* a* a

w = a b b a a b a

match?



Start state



End state

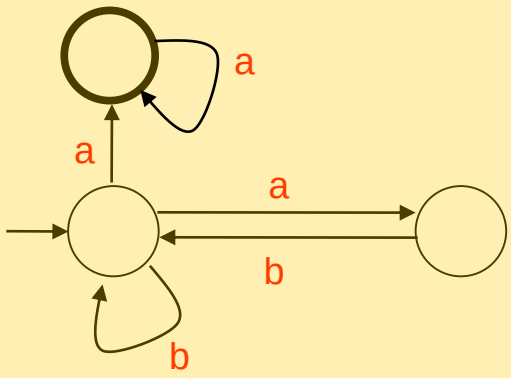
How can you **implement** a regular expression?

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

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

w = a b b a a b a

➔Construct a Finite-State Automaton



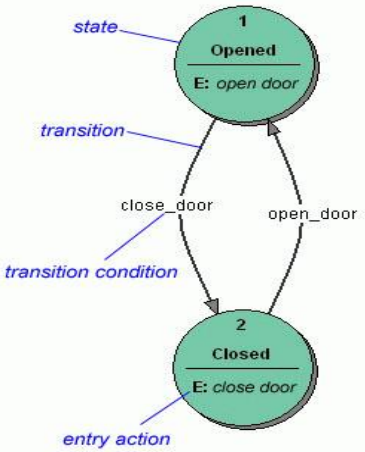
Finite-State Automata (FA) even more useful concept!

→ *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)

→ useful in many areas of CS (verification, compilers, learning, hardware, linguistics, UML, etc)



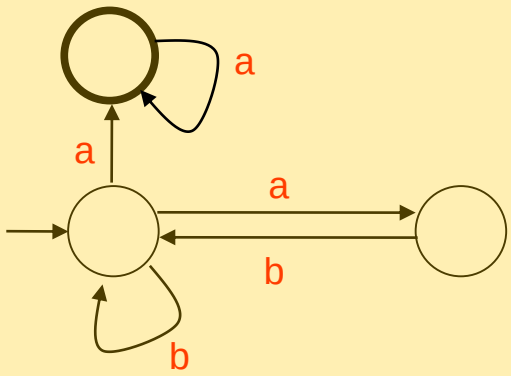
How can you **implement** a regular expression?

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

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

w = a b b a a b a

→Construct a Finite-State Automaton



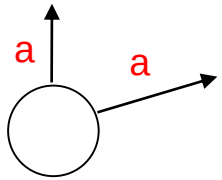
Finite-State Automata (FA)

→ *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(|w|)
"only one finger needed"

FA Matching: time O(|FA| * |w|)
"only at most #states many fingers needed"

→ every FA can be effectively transformed into an equivalent DFA.

→ can take exponential time!

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

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

deterministic FA: run on w takes time **linear** in n = length(w)

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

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

Total Running time O(n + 2^m)

or O(nm)

END
Lecture 2