Applied Databases

Lecture 2
Document Type Definitions (DTDs)

Sebastian Maneth

University of Edinburgh - January 19th, 2017

XML vs JSON

→ XML has advantages over JSON as a **markup language**
→ JSON is **less verbose** than XML & naturally represents objects/arrays

---

Adoption (2016)

Major public APIs using XML only:	Amazon Product Advertisting API
JSON only:	Facebook Graph API, Google Maps API, Twitter API, Pinterest API
both:	Google Cloud Storage, LinkedIn API, Flickr API
Major desktop sofware using XML only:	Microsoft Word, Apache OpenOffice, LibreOffice
Major DBs supporting XML only:	IBM DB2, Microsoft SQL Server, MySQL
JSON only:	MongoDB, CouchDB, OrientDB, Riak
both:	Oracle Database, PostgreSQL, MarkLogic, BaseX, eXistDB, MarkLogic

Accessed 17-Jan-2017 from `http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html`

Recap

XML → widely adopted data exchange format

→ lingua franca for data on the web

→ ordinary text files

→ describe tree structures (indeed graph structures through ID/IDREF)

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

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 the "<"-character!

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]
```

Well-Formed XML

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

→ `<a><b/></a>` `<b/>` is a shorthand for `<b></b>`

→ `<a><b/</a>`

→ `<a></a><b></b>`

→ `<<a>><</a>>`

→ `<a h="17", h="17"></a>`

→ `<a><b><a/></b>`

→ `<a><b></a></b>`

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 (e.g. det. cf. grammars)

O(n^3) parsing time

O(n) time: LR-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)
More powerful than DTD/Schemas wrt tree structure definition

Schematron 2016 (ISO / IEC standard)
Even more powerful rule-based pattern language

Outline of Today: DTDs

DTDs = Document Type Definitions

1. Entity declarations

```
<!ENTITY hi “Hello”>
<greeting>&hi; World!</greeting>
```

2. Attribute-list declarations

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

3. Element type declarations

```
<!ELEMENT friend (name,affil*,email*)>
```

4. Notation declarations
   (not covered)

```
<!NOTATION jpg PUBLIC "JPG 1.0"
                      "image/jpeg">
```

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:

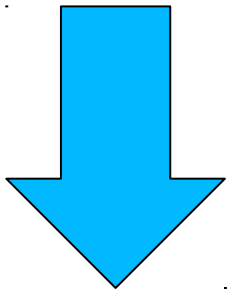
- normalize attribute values (prune white-space etc.) ← if the attribute type is not CDATA
- handle internal entity references (e.g. expand `&hi;` in `greeting`)
- insert default attribute values (e.g. insert `style="small"` in `greeting`)

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

(also next 4 slides)

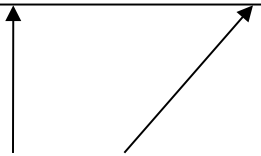
DTDs

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



XML Processor

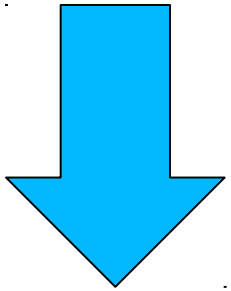
```
<?xml version="1.0"?>
<greeting style="small"> Hello World! </greeting>
```



inserted by Processor

1. Entity Declarations

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

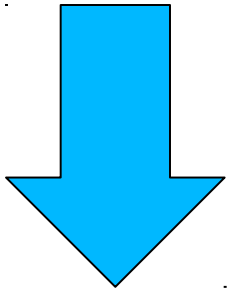


XML Processor

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

1. Entity Declarations

```
<?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> ..
                              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`

1. Entity Declarations

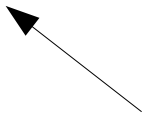
External general parsed entities

→ 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.

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



Is this allowed?

1. Entity Declarations

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

= foo.xml

Try it yourself: (e.g., using the VirtualBox Image of Assignment 1)

```
$ xsltproc id.xslt foo.xml
```

(search for xslt identity transformation to find a source for id.xslt)

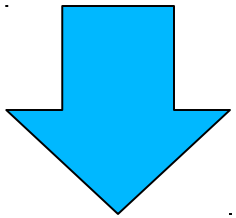
```
$ javac MyDOM.java
$ java MyDOM foo.xml

$ javac MySAX.java
$ java MySAX fool.xml
```

1. Entity Declarations

Entity Expansion

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

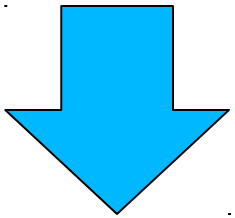


```
<?xml version="1.0"?>
<greeting> HelloHello World! </greeting>
```

1. Entity Declarations

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 "&hi9;&hi9;">
]>
<greeting> &hi10; World! </greeting>
```



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

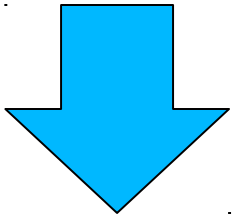
1024 times

1. Entity Declarations

Entity Expansion

```
<?xml version="1.0"?>
<!DOCTYPE greeting [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
...
  <!ENTITY hi10 "&hi9;&hi9;">
]>
<greeting> &hi10; World! </greeting>
```

Try this with `xsltproc` (strange things)



exponential size increase

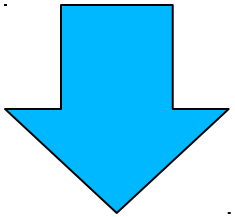
```
<?xml version="1.0"?>
<greeting> Hello. . . Hello World! </greeting>
```

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

1. Entity Declarations

Entity Expansion

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

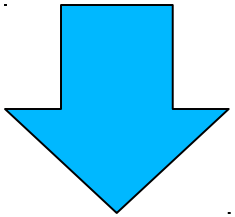


?

1. Entity Declarations

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]

1. Entity Declarations

```
$ cat a.xml
<!DOCTYPE root [
  <!ENTITY hi "Hello">
  <!ENTITY hi1 "&hi;&hi;">
  <!ENTITY hi2 "&hi1;&hi1;">
  <!ENTITY hi3 "&hi2;&hi2;">
  <!ENTITY hi4 "&hi3;&hi3;">
]>
<a><b/> &hi4; <c/></a>

$ xsltproc id.xslt a.xml
Entity: line 1: parser error : Detected an entity reference loop
&hi3;&hi3;
     ^
a.xml:8: parser error : Detected an entity reference loop
<a><b/> &hi4; <c/></a>
             ^
unable to parse a.xml

$
```

→ xsltproc / java DOM&SAX
not accurate wrt DTD validation!

→ try an online checker or a better tool!

XML Entity Expansion

Attack subtypes

There are three attack subtypes:

1. **XML Generic Entity Expansion** <br\> XML Generic Entity Expansion is the most simple attack. All the attacker has to do is declare an entity with over long content and use the entity many times in the SOAP Message. When parsing the SOAP message all entities are resolved which causes an exhaustion of the RAM of the attacked web service. <br\>As a rough estimate the attack works when using the following parameters: <br\>-**Length of string:** more than 10^5 characters <br\>-**Number of entity occurrences in document:** more than 30,000 occurrences. <br\><br\>For more details please refer to the work of Leroy Metin Yaylacioglu listed in the reference section. <br\>This attack is also known as the "Quadritiv Blowup DOS Attack"<br\><br\>
1. **XML Recursive Entity Expansion** <br\>The basic idea behind attack is the same as the XML Generic Entity Expansion attack,however the attack is a little more elegant. When successful a relatively small SOAP Message is expanded to a large memory structure which exhausts the application's RAM. <br\>Lets explain the attack based on an example: <br\>The attacker starts off by defining at least 100 entities named x0 to x100. The entity &x0; gets a fixed value assigned. All other entities &x1; through &x100; contain the entity name of the previous entity **twice** as a value. If we define the entity x50, then the value of the entity is "&x49;&x50;". Later in the document the &x100; is used exactly once. This is sufficient to devastate the web services availability since the document grows exponentially. With every recursion the number of entities in the document doubles, resulting in 2^101 repetitions of the value of the entity &x0;. <br\>This attack is also known as "XML Bomb"<br\><br\>
2. **XML Remote Entity Expansion** <br\> When using the XML Remote Entity Expansion attack an attacker defines an external entity, that in return also points to an external entity and so on. Before doing any further processing the parser has to retrieve all external entity definitions. Depending on the web service load this can use up all the remaining system resources of the web service and therefore render it unavailable. <br\><br\>
3. **XML C14N Entity Expansion** <br\> The attack works just as described above. The only difference is

Main page
Get involved
Contact
About

Web Service Attacks
Attack Structure
Attacks by Category

Security Evaluation
Test Environment
Pentest Tools
Open Pentests

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

2. Attribute-List Declarations

- `<!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).

2. Attribute-List Declarations

(1) Fixed default attribute value

Syntax:

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

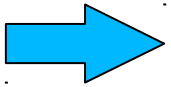
DTD example:

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

XML examples:

```
<sender company="Microsoft">
```

```
<sender>
```



```
<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.

2. Attribute-List Declarations

(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=“bank transfer”>
```

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

2. Attribute-List Declarations

(2b) Enumerated attribute type

Syntax:

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

DTD example:

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

XML example:

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

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

2. Attribute-List Declarations

(3) Required attribute

Syntax:

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

DTD example:

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

XML example:

```
<person securityNumber="31415926">
```

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.

2. Attribute-List Declarations

(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.

3. Element Type Declarations

Syntax:
`<!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: `...+`

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

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

Note

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

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

- 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 text node here

Regular Expressions

- zero or more: `...*`

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

valid document

→ is it valid?

Regular Expressions

- zero or more: `...*`

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

not valid

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 ubiquitous in computer science!

→ used in EBNF, for defining the syntax of PLs

→ used in various unix tools (e.g., `grep, ed`)

→ supported in most PLs (esp. `Perl`), text editors

→ classical concept in CS (Stephen Kleene, 1950's)

Today
→ heavily used for network intrusion detection
→ expressions can be quite large

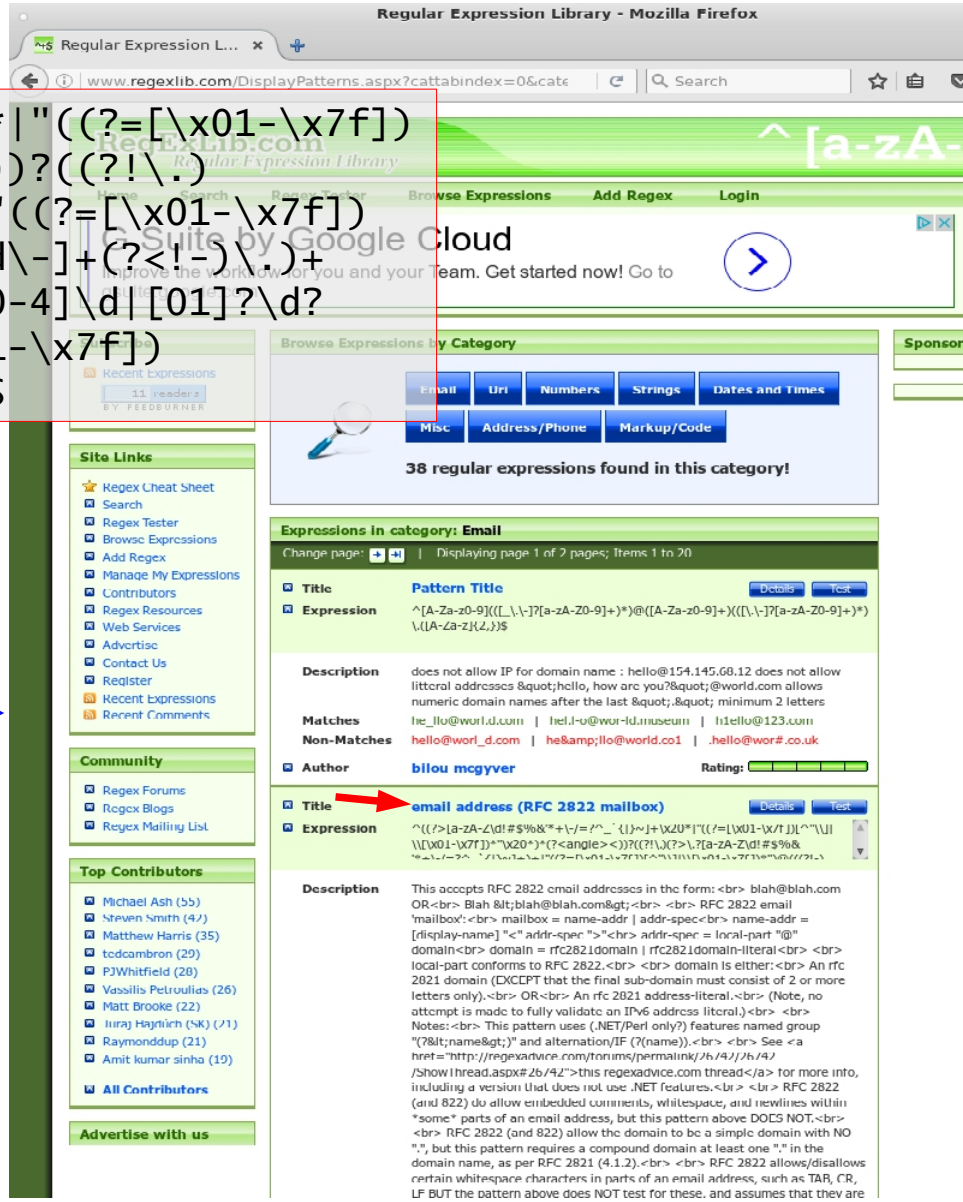
```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*"\x20*)*(?<angle><))?((?!\.)
(?>\.?[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+
[a-zA-Z]{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])
[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

Today

→ heavily used for network intrusion detection

→ expressions can be quite large

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*"\x20*)*(?<angle><))?((?!\.)
(?>\.?[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+)+|"((?=[\x01-\x7f])
[^"\\]|\\[\x01-\x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+
[a-zA-Z]{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])
[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```



Matches email addresses, e.g.

```
name.surname@blah.com
Name Surname <name.surname@blah.com>
"b. blah"@blah.co.nz
```

Accessed 19-Jan-2017 from `http://www.regexlib.com`

Regular Expressions are ubiquitous in computer science!

→ used in EBNF, for defining the syntax of PLs

→ used in various unix tools (e.g., `grep, ed`)

→ supported in most PLs (esp. `Perl`), text editors

→ classical concept in CS (Stephen Kleene, 1950's)

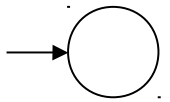
How can you **implement** a regular expression?

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

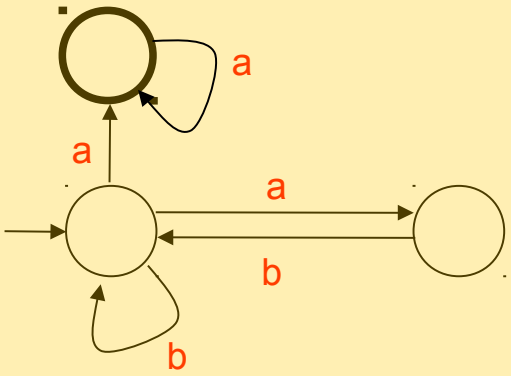
How can you **implement** a regular expression?

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



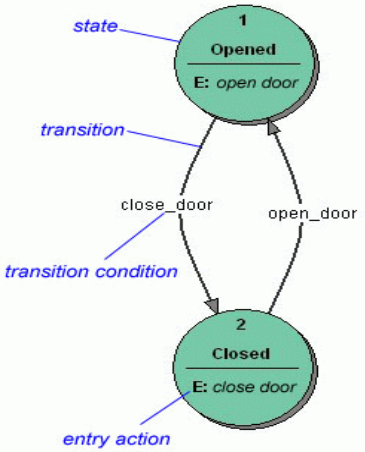
Finite-State Automata (FA) even more useful concept!

→ *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)



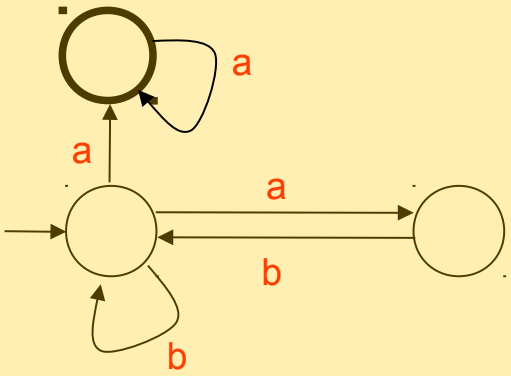
How can you **implement** a regular expression?

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



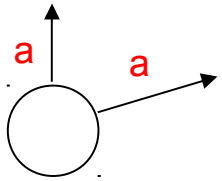
Finite-State Automata (FA)

→ *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