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

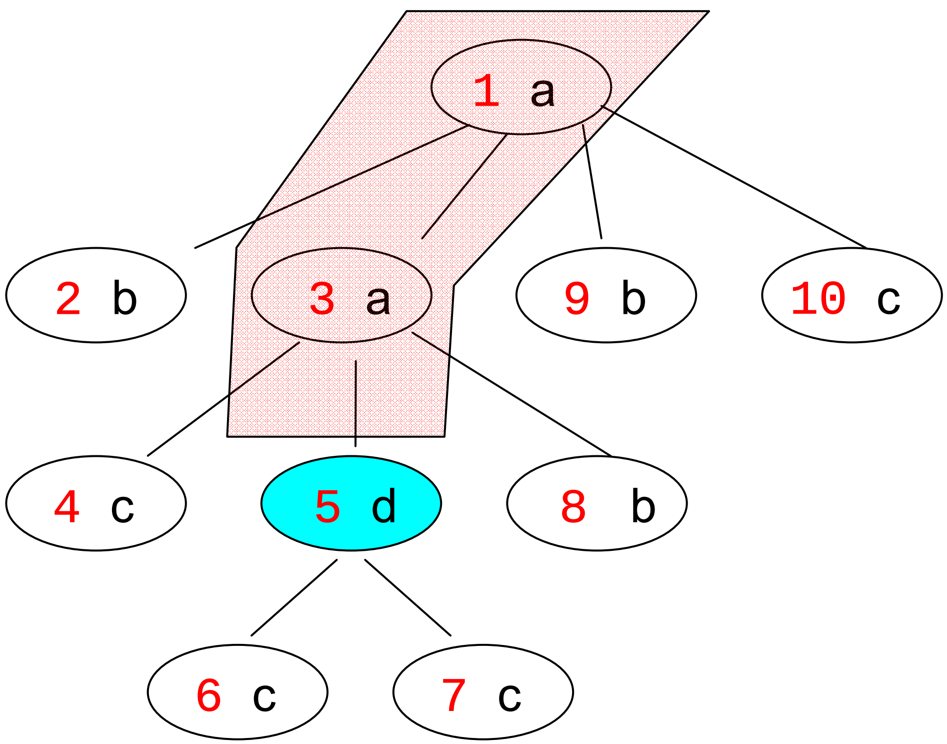
---

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}

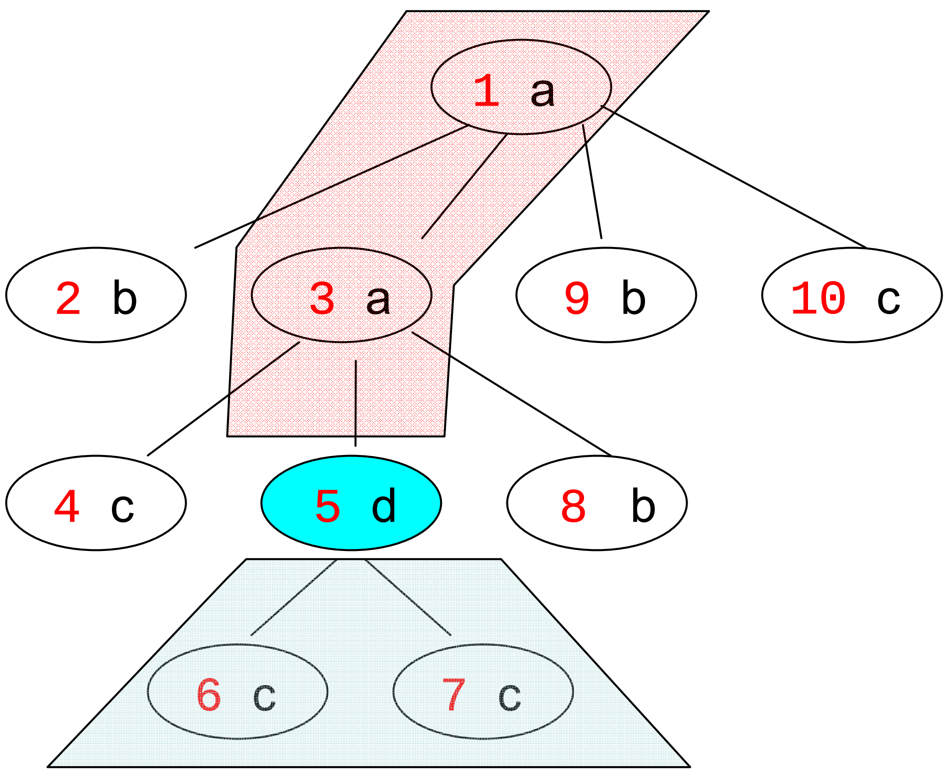


ancestor(5) = { 1, 3 }

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}



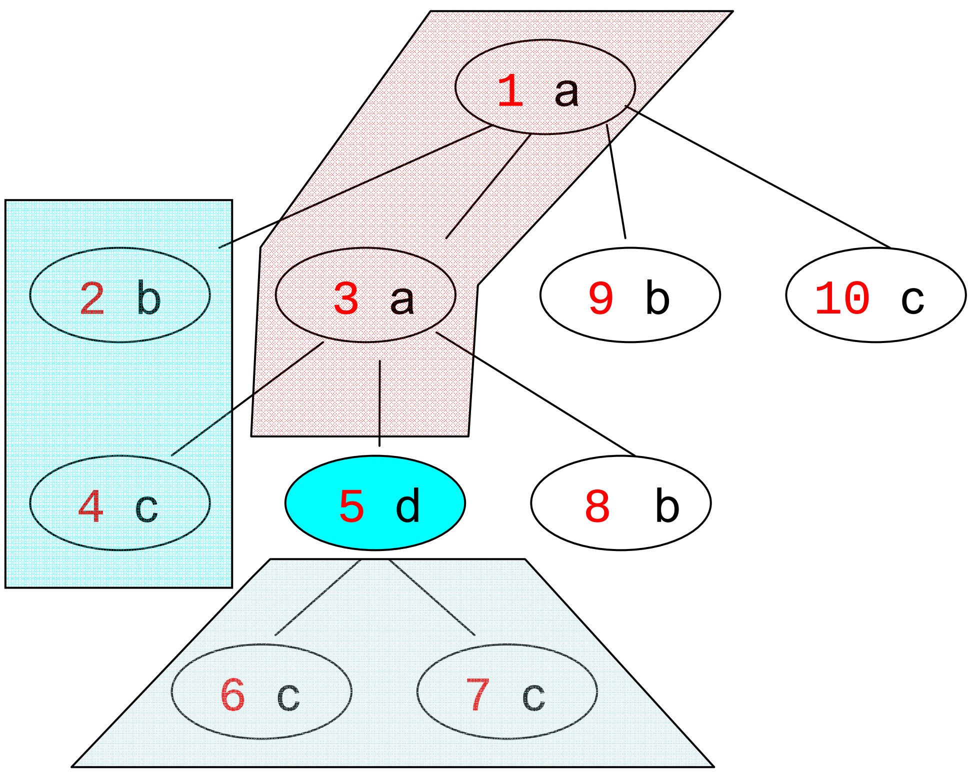
ancestor(5) = { 1, 3 }

descendant(5) = { 6, 7 }

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 }

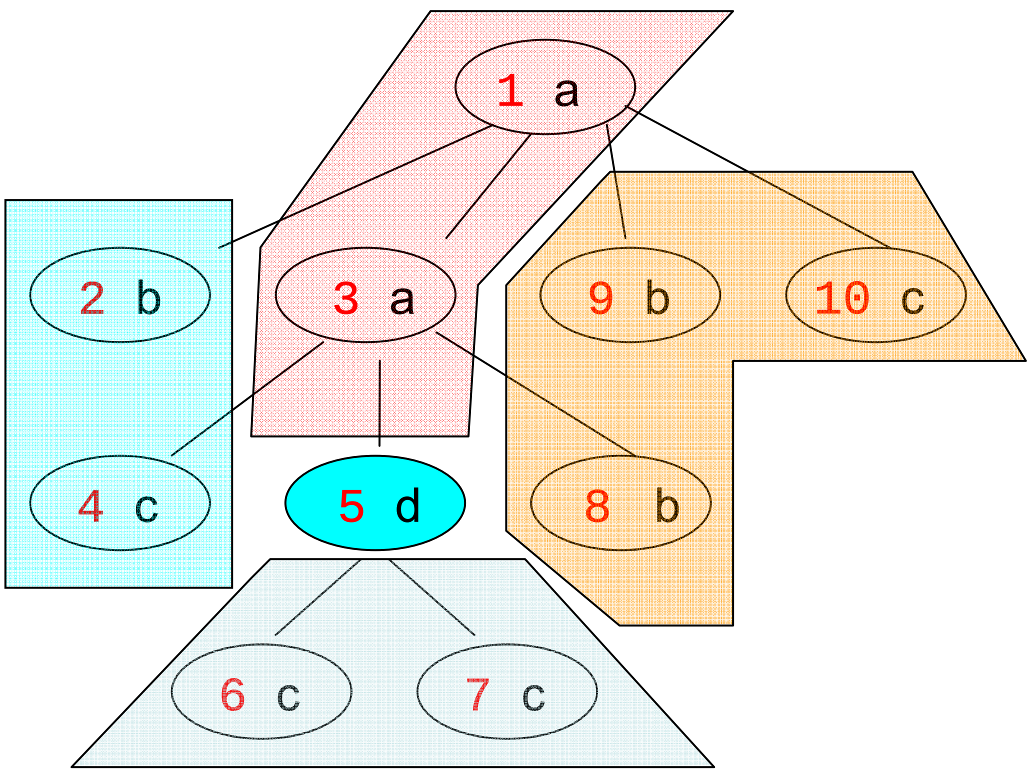
descendant(5) = { 6, 7 }

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 }

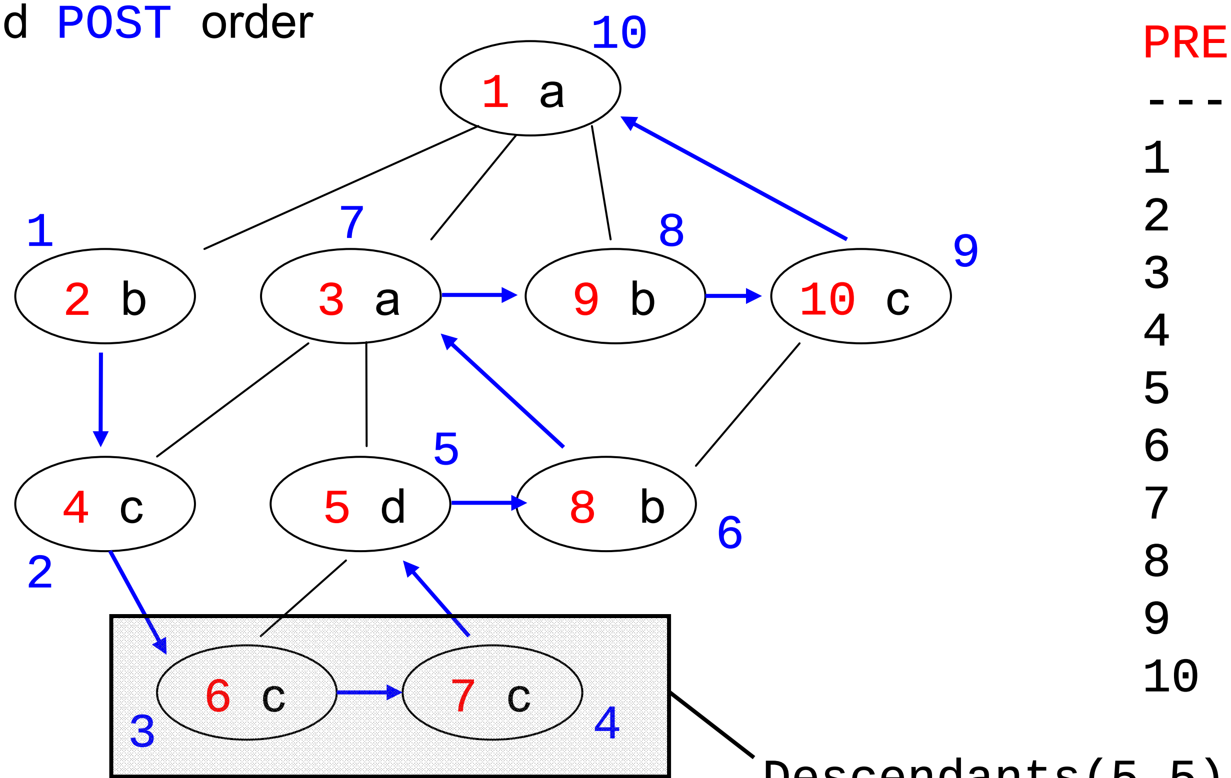
preceding(5) = { 2, 4 }

following(5) = { 8, 9, 10 }

self(5) = { 5 }

Pre/Post Encoding

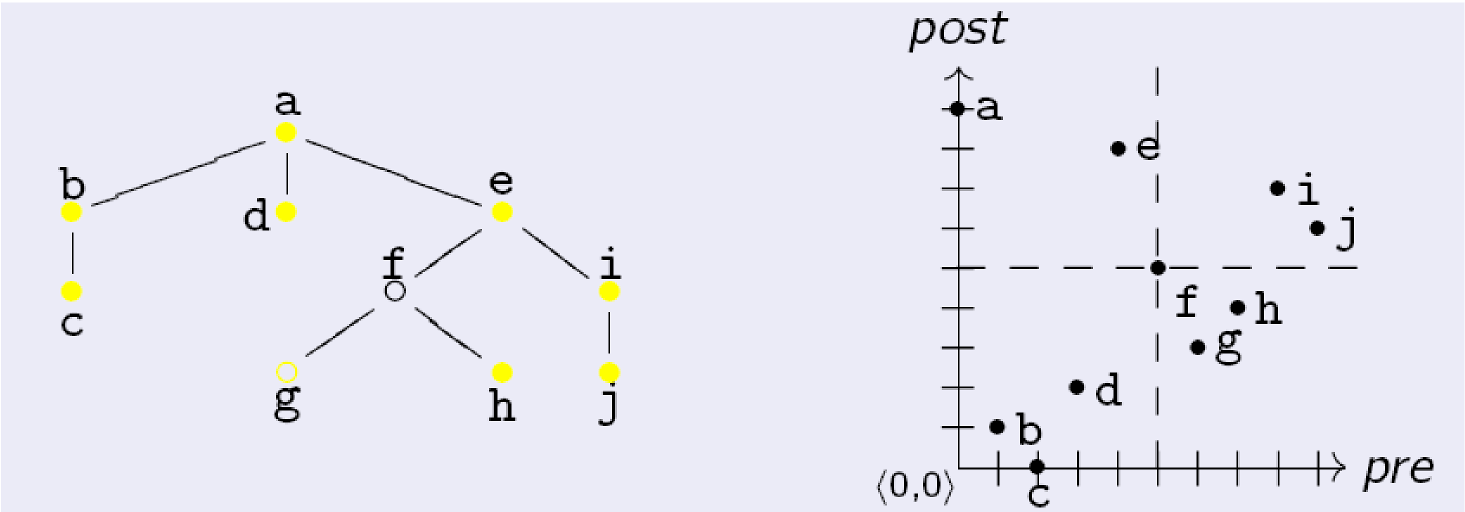
➔ Add POST order

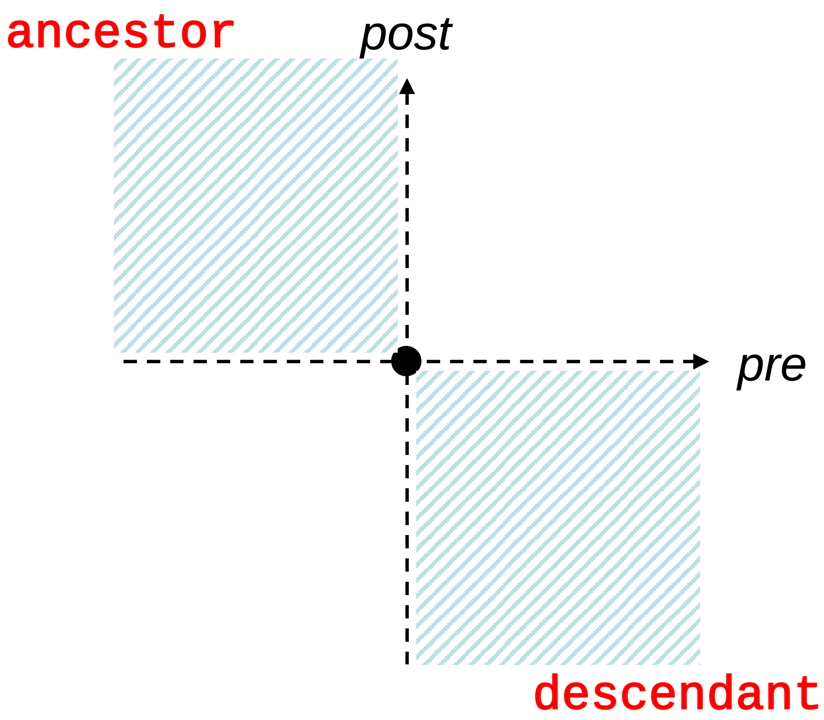


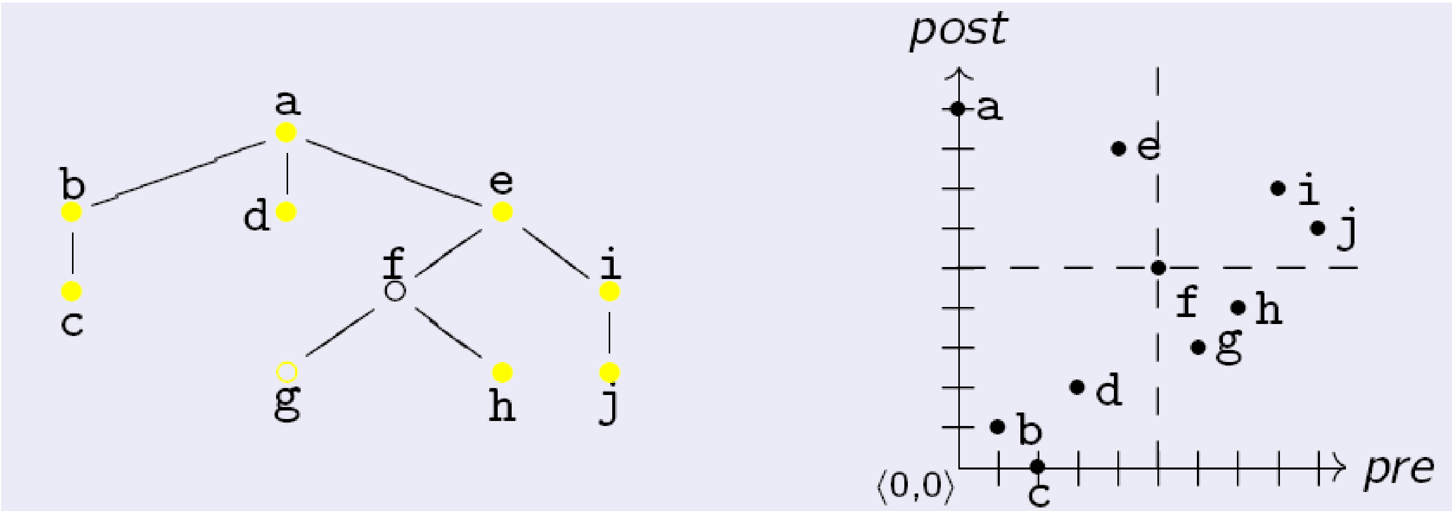
PRE	POST	lab
1	10	a
2	1	b
3	7	a
4	2	c
5	5	d
6	3	c
7	4	c
8	6	b
9	8	b
10	9	c

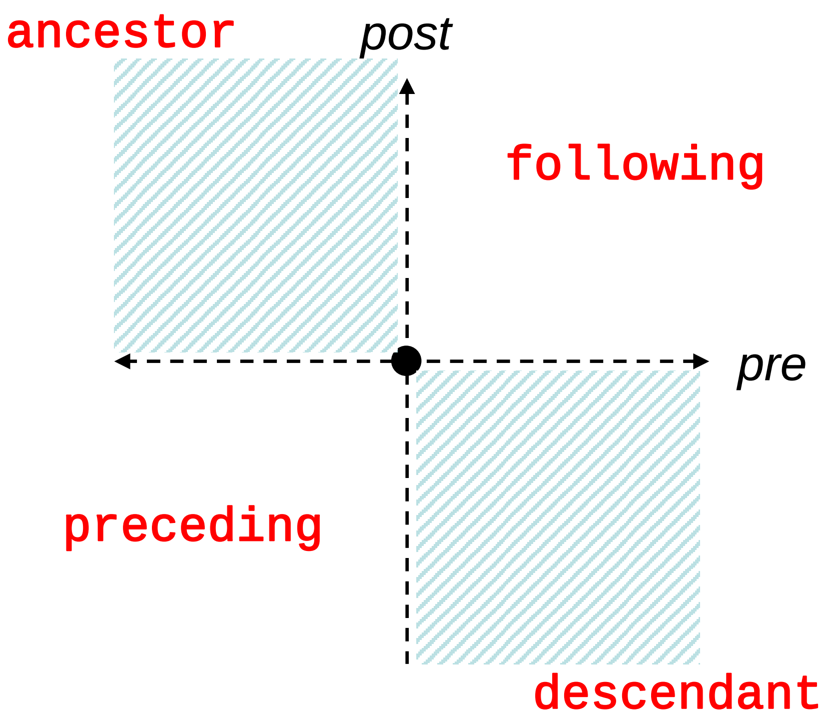
```
Descendants( Pre, Post ) =

  SELECT r1.pre FROM DOCtable r1,
              WHERE r1.pre > Pre
                AND r1.post < Post
```

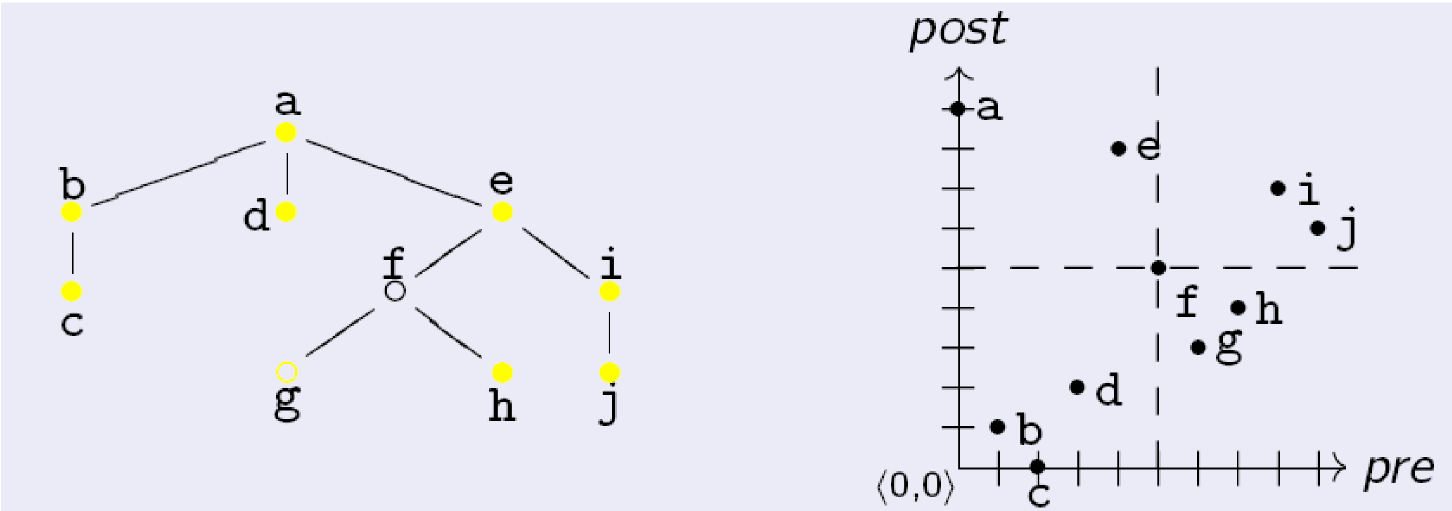
“structural join”

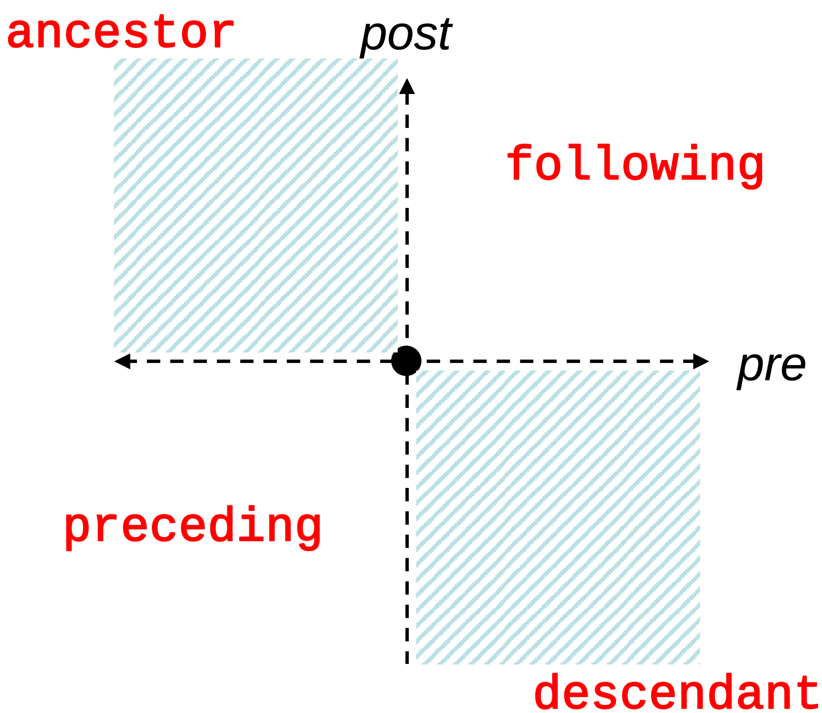

a
b
d
e
c
f
i
g
h
j
post
a
e
i
j
f
h
g
d
b
⟨0,0⟩
c
pre


ancestor
post
pre
descendant

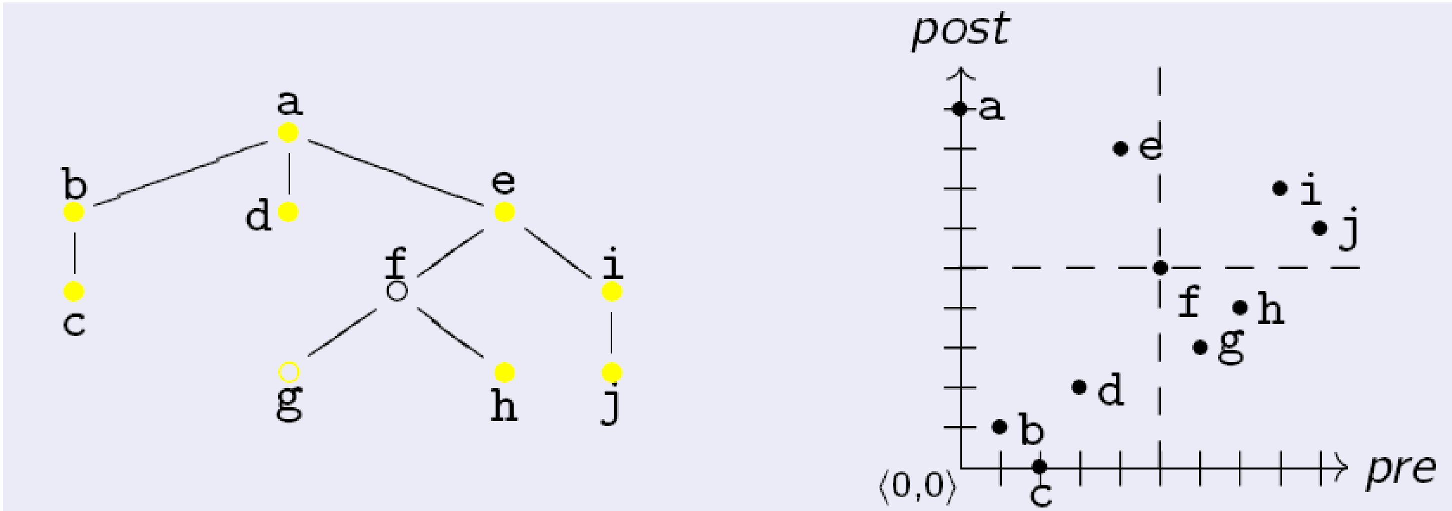

a
b
c
d
e
f
g
h
i
j
post
pre
⟨0,0⟩

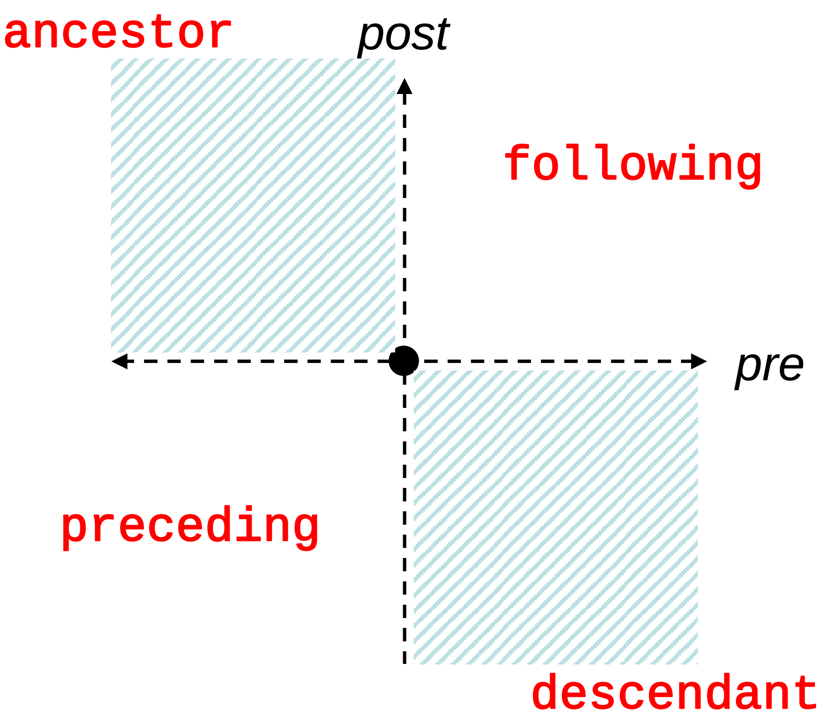

ancestor
post
following
pre
preceding
descendant





firstChild(pr, po) = ?

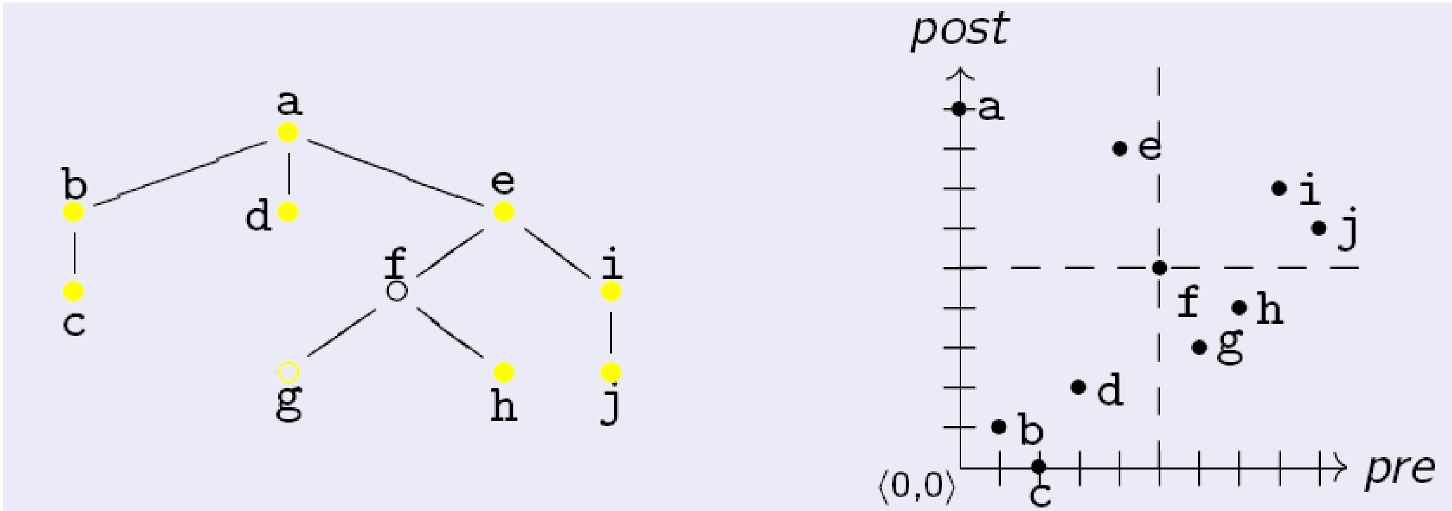


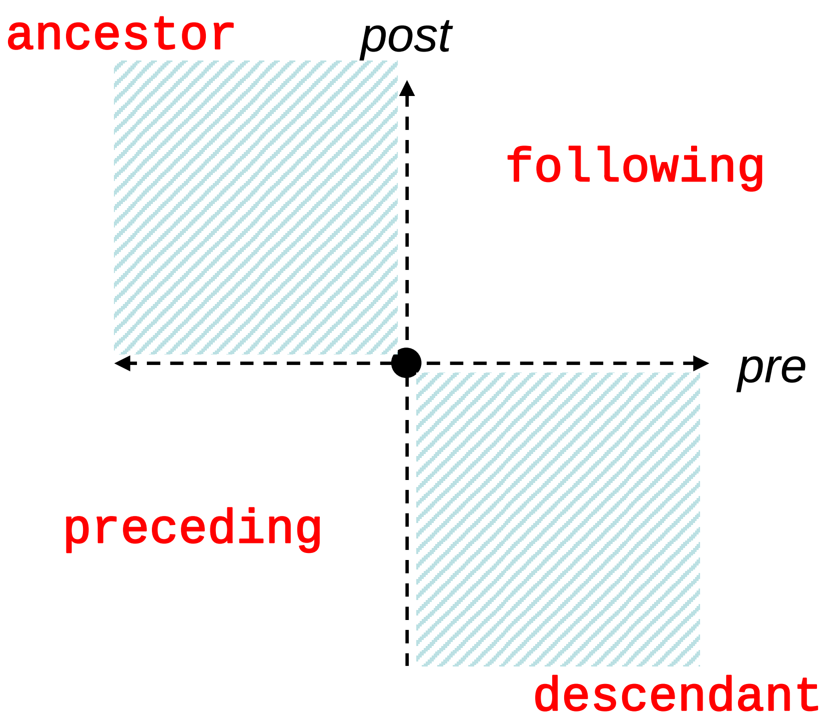


`firstChild`(pr, po) = left-most node, below and to the right of (pr,po)

or, equivalently

node (pr+1, p) with p < po, if it exists.





firstChild(pr, po) = left-most node,
below and to the right of (pr,po)

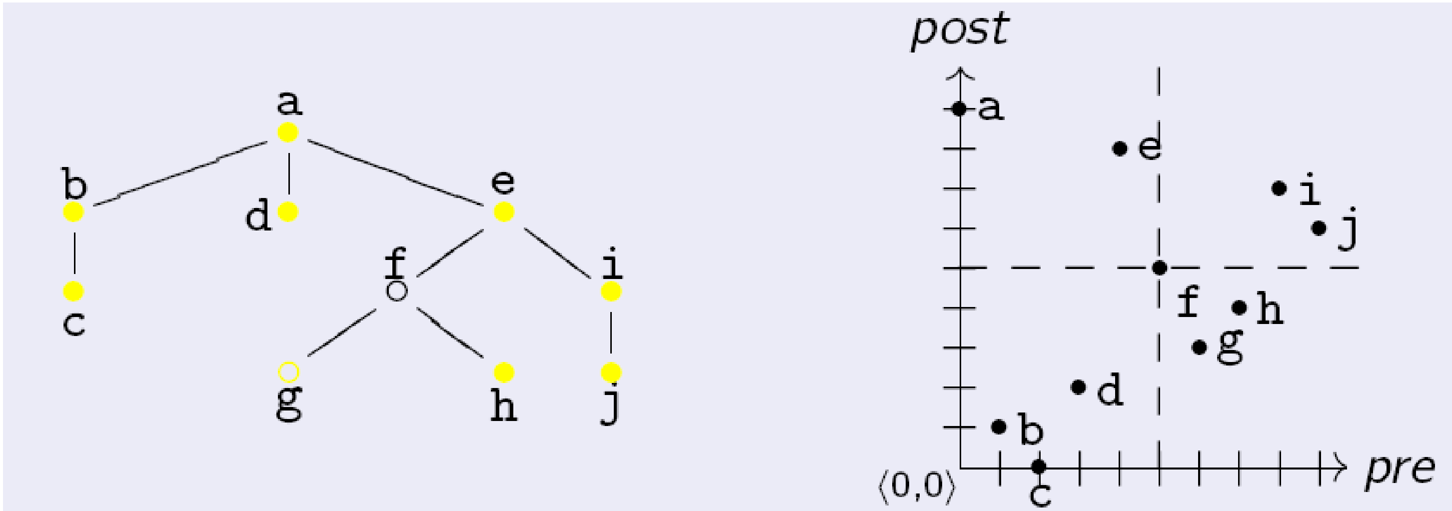
or, equivalently

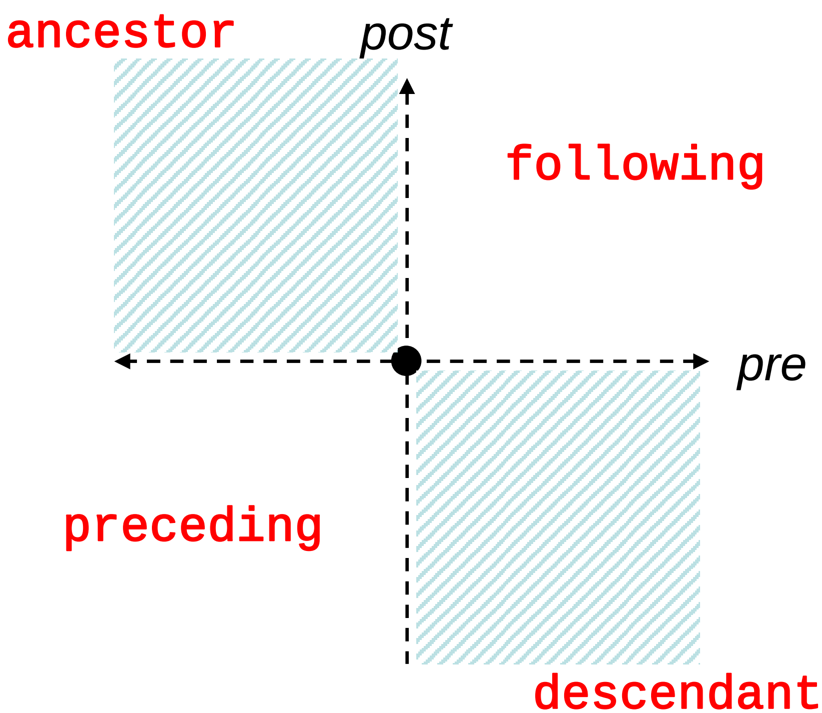
node (pr+1, p) with p < po, if it exists.

---

lastChild(pr, po) =

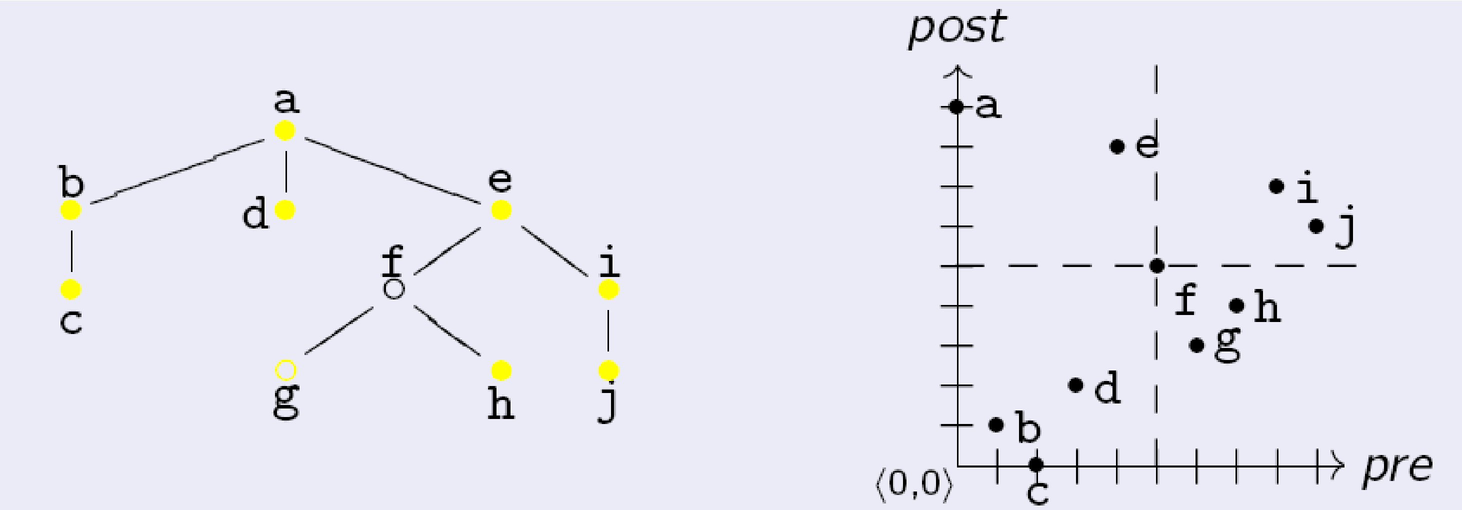
node (p, po-1) with p > pr, if it exists.

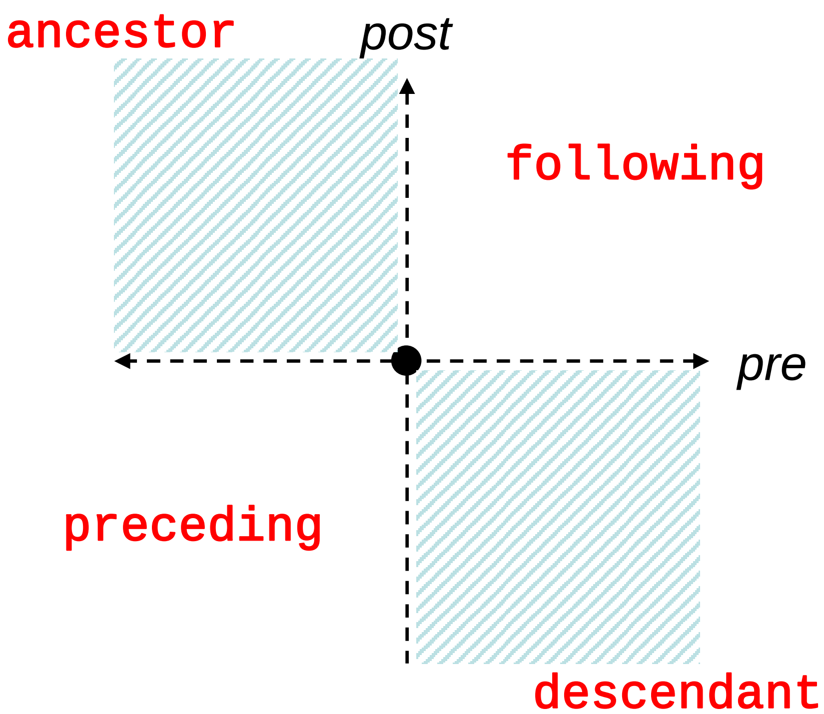




firstChild(pr, po) = left-most node,
below and to the right of (pr,po)

nextSibling(pr, po) =
left-most node,
→ to the right
→ up
such that …?





firstChild(pr, po) = left-most node,
below and to the right of (pr,po)

nextSibling(pr, po) =
left-most node (pr2, po2),
→ to the right
→ up
such that there is no node
with post value > po and < po2
to the left.

e.g., **not** c- and d-node
(because b-node is inbetween..)

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?

`firstChild`(pr, po) = left-most node, below and to the right of (pr,po)

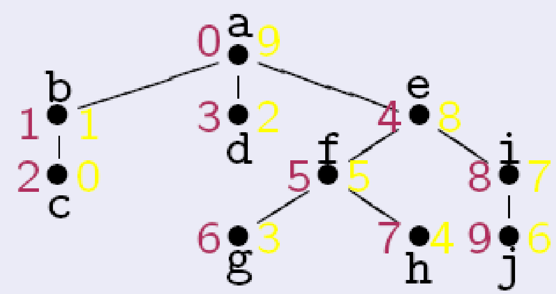
`nextSibling`(pr, po) =
left-most node (pr2, po2),
→ to the right
→ up
such that there is no node with post value > po and < po2 to the left.

e.g., **not** c- and d-node
(because b-node is inbetween..)

XPath Accelerator encoding

XML fragment *f* and its skeleton tree

```
<a>
  <b>c</b>
  <!--d-->
  <e><f><g/><?h?></f>
     <i>j</i>
  </e>
</a>
```



Pre/post encoding of *f*: table `accel`

pre	*post*	*par*	*kind*	*tag*	*text*
0	9	NULL	elem	a	NULL
1	1	0	elem	b	NULL
2	0	1	text	NULL	c
3	2	0	com	NULL	d
4	8	0	elem	e	NULL
5	5	4	elem	f	NULL
6	3	5	elem	g	NULL
7	4	5	pi	NULL	h
8	7	4	elem	i	NULL
9	6	8	text	NULL	j

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

---

pre	post	~~par~~	~~kind~~	tag	text
0	9	~~NULL~~	~~elem~~	a	NULL
1	1	~~0~~	~~elem~~	b	NULL
2	0	~~1~~	~~text~~	NULL	c
3	2	~~0~~	~~com~~	NULL	d
4	8	~~0~~	~~elem~~	e	NULL
5	5	~~4~~	~~elem~~	f	NULL
6	3	~~5~~	~~elem~~	g	NULL
7	4	~~5~~	~~pi~~	NULL	h
8	7	~~4~~	~~elem~~	i	NULL
9	6	~~8~~	~~text~~	NULL	j

→ 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/LEVEL encoding to the DB (e.g., MySQL)
- → sends the queries to the DB
- → receives the answers and prints/evaluates them

---

pre	post	par	kind	tag	text
0	9	NULL	elem	a	NULL
1	1	0	elem	b	NULL
2	0	1	text	NULL	c
3	2	0	com	NULL	d
4	8	0	elem	e	NULL
5	5	4	elem	f	NULL
6	3	5	elem	g	NULL
7	4	5	pi	NULL	h
8	7	4	elem	i	NULL
9	6	8	text	NULL	j

→ Only element/text nodes!

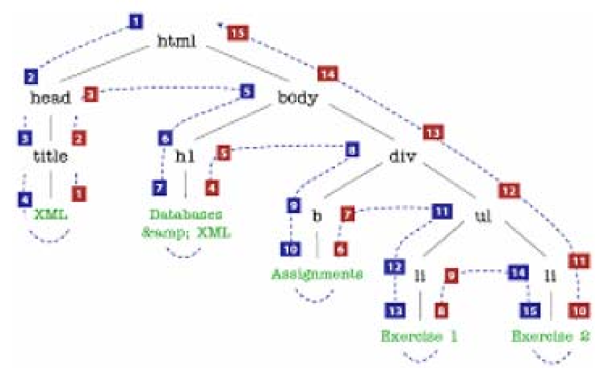
PLUS **attributes**

```
<a color="green">
...
</a>
```

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

XML Database – Table Storage

Pre/Post Plane:




```
<html>
   <head>
      <title>XML</title>
   </head>
   <body bgcolor="#FFFFFF" text = "#000000">
      <h1>Databases &amp; XML</h1>
      <div align="right">
         <b>Assignments</b>
         <ul>
            <li>Exercise 1</li>
            <li>Exercise 2</li>
         </ul>
      </div>
   </body>
</html>
```

Assignment 3 Generate (pre,post,tag,text)-table

```
<a>
  <b>Hello World</b>
  <c></c>
</a>
```

```
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
```

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**

```
<a>
  <b>Hello World</b>
  <c></c>
</a>
```

```
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
```

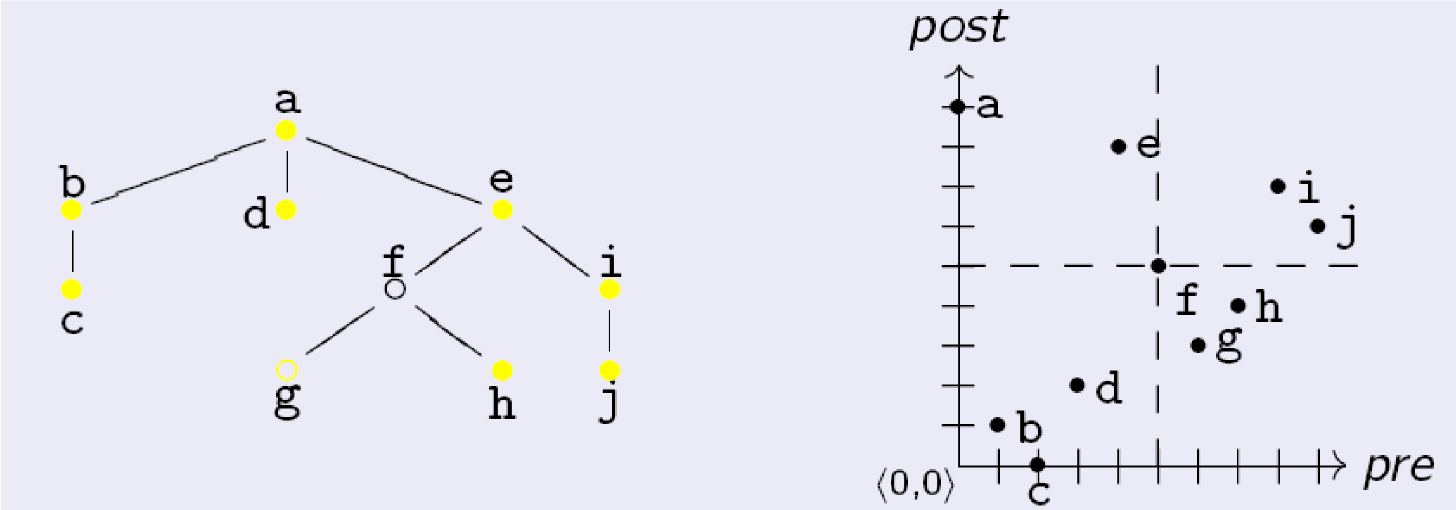
from the document, generate SQL insert statements

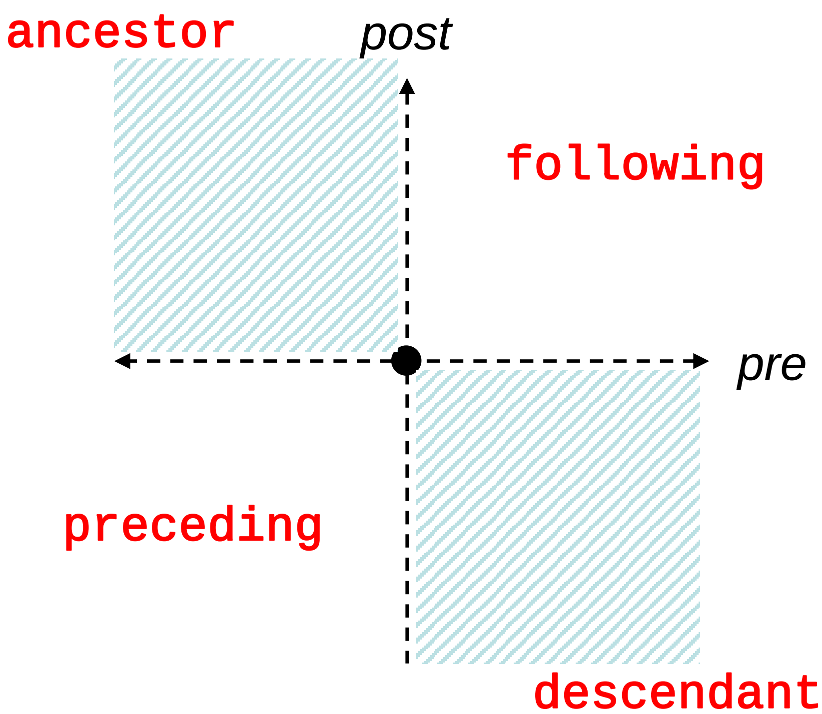
```
INSERT INTO book_tbl (pre,post,tag,text)
VALUE (1, 12, "book", null);
```

```
<a>
  <b>Hello World</b>
  <c a1="123"></c>
</a>
```

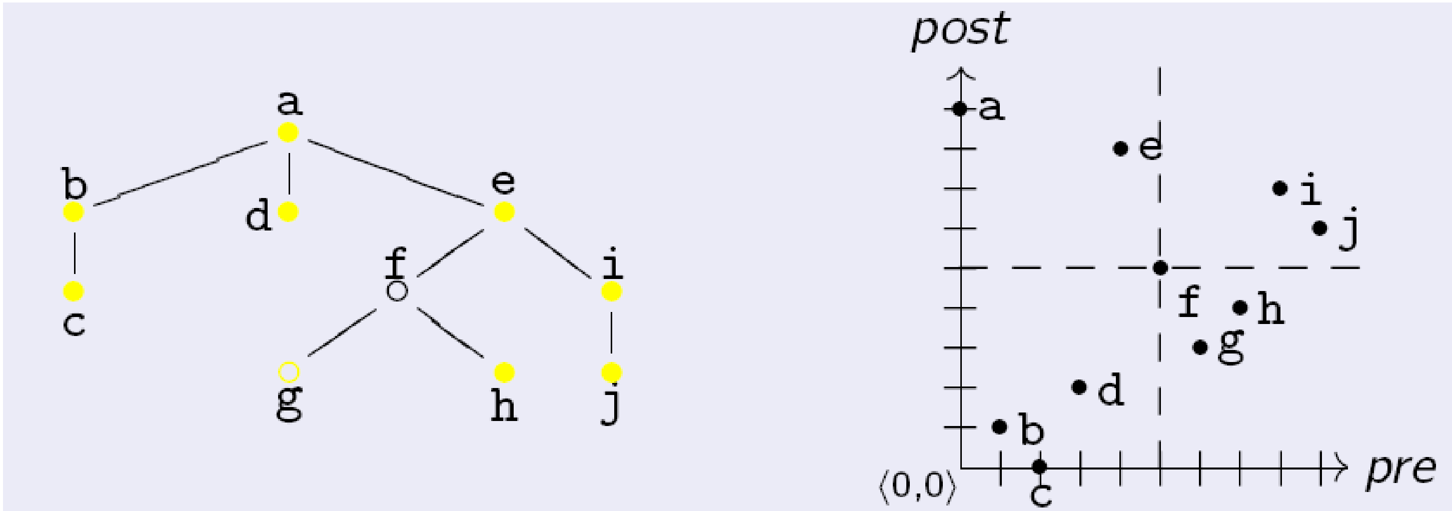
```
pre | attr | value
------------------
  4 |  a1  | "123"
```

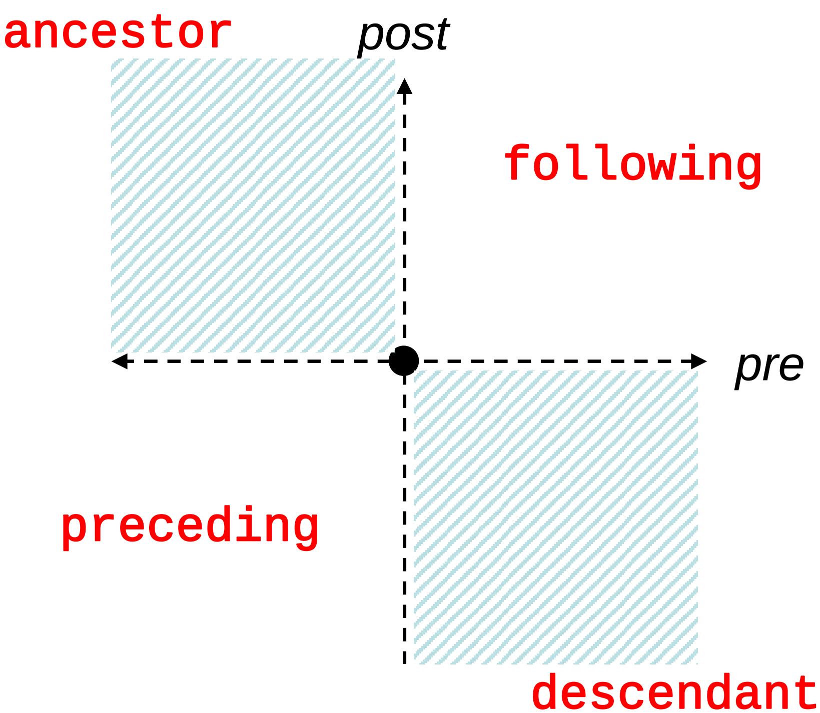
```
INSERT INTO book_tbl (pre,post,tag,text)
VALUE (1, 12, "book", null);
```





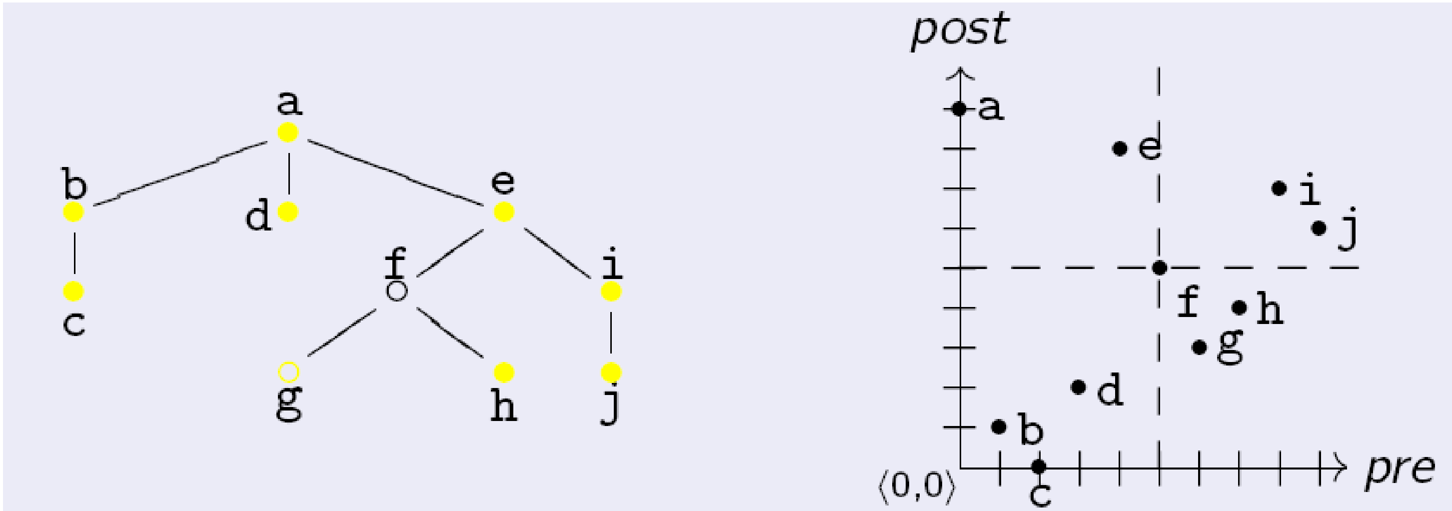
nextSibling(pr, po, LE) =
left-most node (pr2, po2, LE2),
→ to the right
→ up
~~such that there is no node with post value > po and < po2.~~

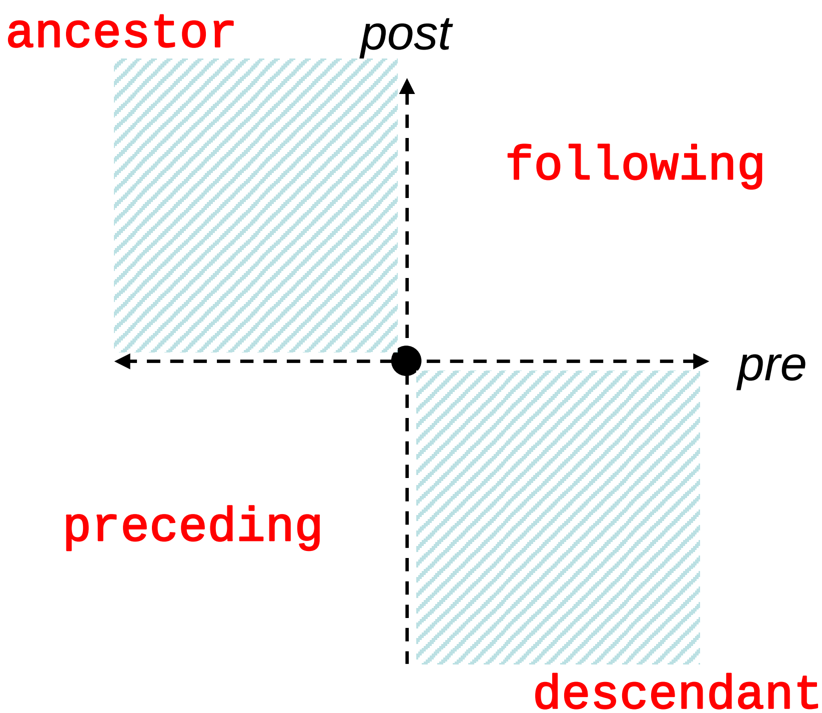




nextSibling(pr, po, LE) =
left-most node (pr2, po2, LE2),
→ to the right
→ up
~~such that there is no node~~
~~with post value > po and < po2.~~

if (LE == LE2)



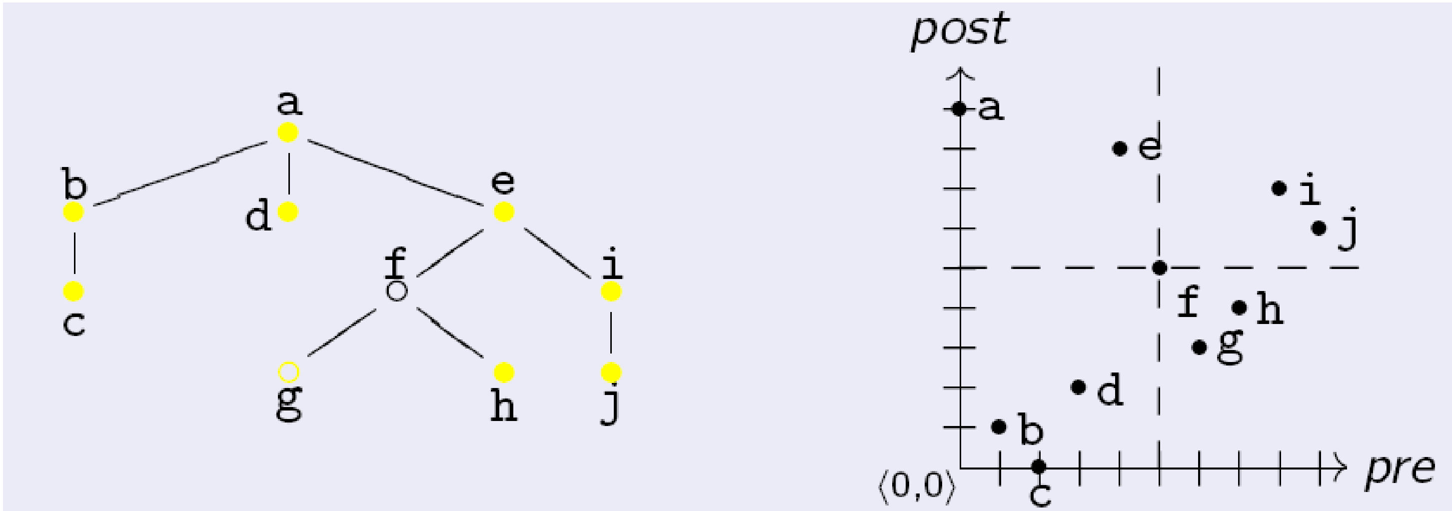


nextSibling(pr, po, LE) =
left-most node (pr2, po2, LE2),
→ to the right
→ up
~~such that there is no node with post value > po and < po2.~~

↑
if (LE == LE2)

---

nextSibling(pr, po, pa) = (pr2, po2, pa)
such that pr<pr2 and there is no
(pr3, po3, pa) with pr<pr3<pr2



Using
(pre, SIZE, LEVEL)-encoding:

→ How to compute
all children of a node (p,s,l)?

→ Can you compute the post value
from given (pre, size, level)?

nextSibling(pr, po, LE) =
left-most node (pr2, po2, LE2),
→ to the right
→ up
~~such that there is no node~~
~~with post value > po and < po2.~~

if (LE == LE2)

---

nextSibling(pr, po, pa) = (pr2, po2, pa)
such that pr<pr2 and there is no
(pr3, po3, pa) with pr<pr3<pr2

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 - 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 - Assignments

1. Read XML, using DOM parser. Create document statistics.
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

Lecture 4

DTDs & Reg. Exprs

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 efficient 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

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 "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)

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>
```

Or:
Store DTD in `gr.dtd`, and use:

```
<!DOCTYPE greeting SYSTEM “gr.dtd”>
```

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)

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`.

```
<!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>
```

This grammatical description has some obvious shortcomings:

- we cannot express that, e.g. **`protein`**, must contain a non-negative number
- **`unit`** should only be allowed when **`amount`** is present
- the **`comment`** element should be allowed to appear anywhere
- nested **`ingredient`** elements should only be allowed when **`amount`** is absent

- **<!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 (value_1|value_2|..) “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

This is a simple subset of SGML DTD.

How??

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

The Definition of Mixed Content

- *Mixed content* is described by a repeatable OR group

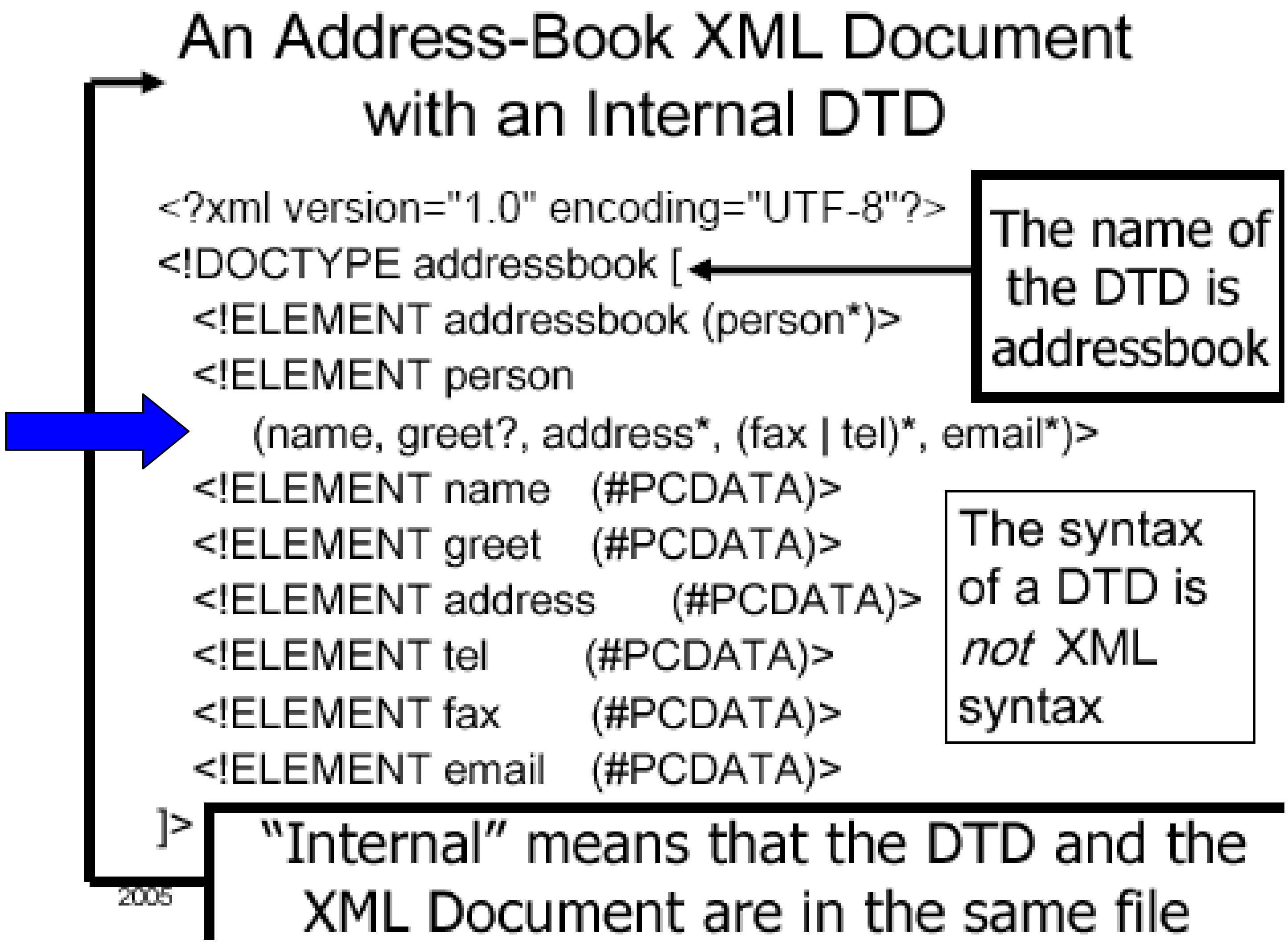
  (#PCDATA | *element-name* | …)*

  – 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!

Most interesting content mode:

Regular Expression



Most interesting content mode:

Regular Expression

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?

Specifying the Structure (cont'd)

• addr*	to specify 0 or more address lines
• tel | fax	a tel *or* a fax element
• (tel | fax)*	0 or more repeats of tel or fax
• email*	0 or more email elements

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
• e1 | 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* a

w = a b b a a b a

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)

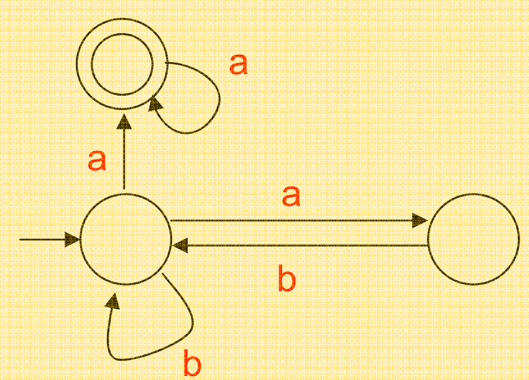
How can you **implement** a regular expression?

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

→Construct a Finite-State Automaton



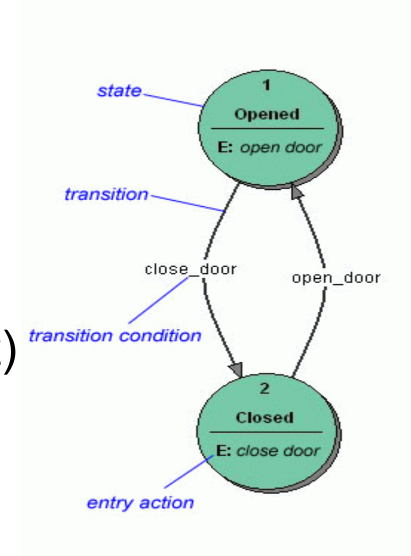
Finite-State Automata (FA) even more useful concept!

→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)



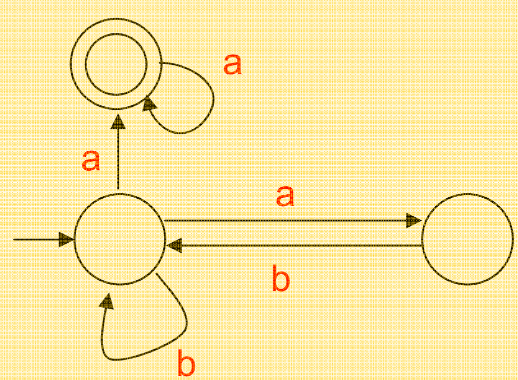
How can you **implement** a regular expression?

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

→Construct a Finite-State Automaton



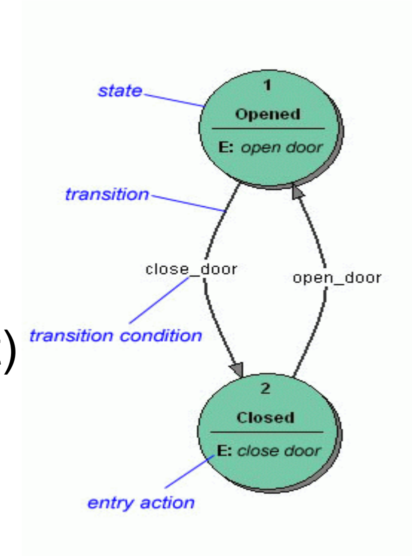
Finite-State Automata (FA) even more useful concept!

→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)

→for every FA there is an equivalent *deterministic* FA
(= per letter *at most one* outgoing edge)



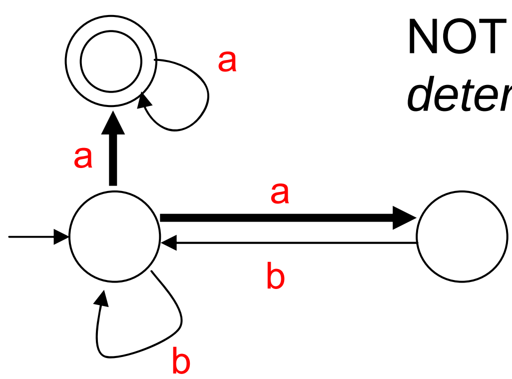
How can you **implement** a regular expression?

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

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

w = a b b a a b a

→Construct a Finite-State Automaton



NOT
deterministic

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

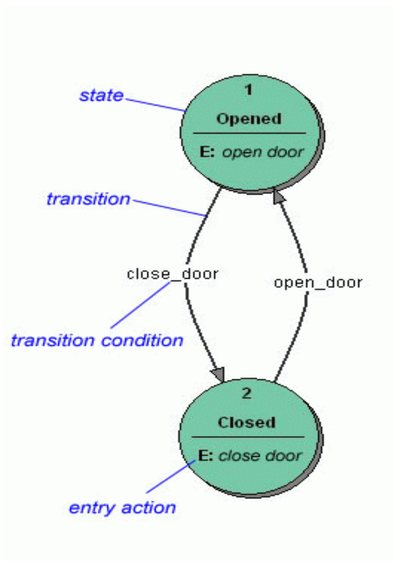
→they **truly** incarnate *constant memory* computation.

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

→for every Reg Exp there is a FA (and vica versa)

→for every FA there is an equivalent *deterministic* FA

(= per letter *at most one* outgoing edge)



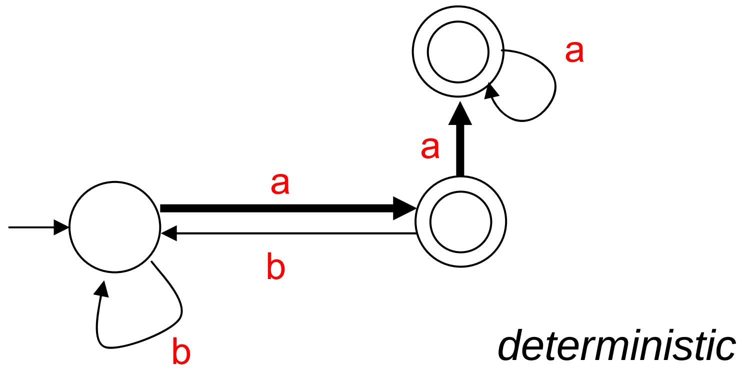
How can you **implement** a regular expression?

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

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

w = a b b a a b a

→Construct a Finite-State Automaton



deterministic

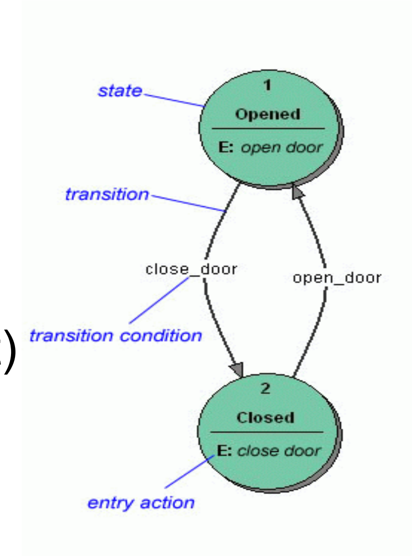
Finite-State Automata (FA) even more useful concept!

→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)

→for every FA there is an equivalent *deterministic* FA
(= per letter *at most one* outgoing edge)

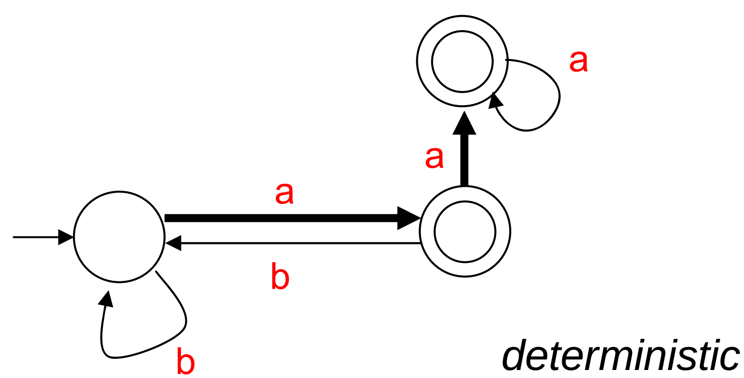


How can you **implement** a regular expression?

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

deterministic FA: run on w takes
time **linear** in length(w)
and **constant space** (#states, e.g., 3 →)

→Construct a Finite-State Automaton



Finite-State Automata (FA)

→ For every FA you can build and equivalent *deterministic* FA ☺
But, could become **exponentially** larger, ☹
sometimes unavoidable (FA is more *succinct*)

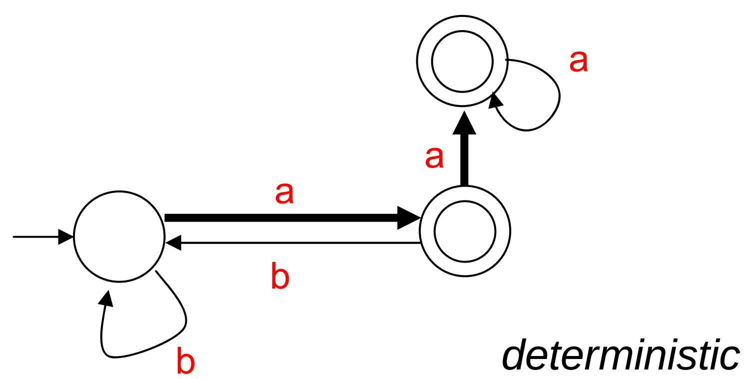
→ For every *deterministic FA* you can build a *minimal unique equivalent* one
Thus, equivalence is decidable! ☺
Very rare! --- E.g., equivalence of EBNF's is NOT decidable.

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(w)
and **constant space** (#states, e.g., 3 →)

→Construct a Finite-State Automaton



Why?
Can you find an example?

Finite-State Automata (FA)

→ For every FA you can build and equivalent *deterministic* FA ☺
But, could become **exponentially** larger, ☹
sometimes unavoidable (FA is more *succinct*)

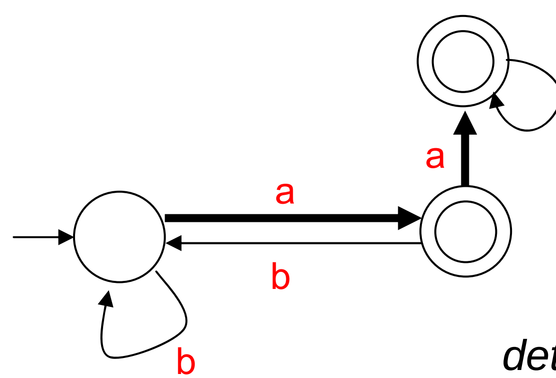
→ For every *deterministic FA* you can build a *minimal unique equivalent* one
Thus, equivalence is decidable! ☺
Very rare! --- E.g., equivalence of EBNF's is NOT decidable.

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(w)
and **constant space** (#states, e.g., 3 →)

→Construct a Finite-State Automaton



deterministic

Finite-State Automata (FA)

→ For every FA you can build and equivalent *deterministic* FA ☺
But, could become **exponentially** larger, ☹
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(w)

Algorithm

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

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

n = length(w) **Total Running time** O(n + 2^m)

Finite-State Automata (FA)

→ For every FA you can build and equivalent *deterministic* FA ☺
But, could become **exponentially** larger, ☹
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(w)

Algorithm

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

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

n = length(w) **Total Running time** O(n + 2^m)

→ Other alternative: O(nm)

To avoid these expensive running times

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

Is small! ☺
size is only O(m)

W3C DTD-defin.

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)

Algorithm

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

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

n = length(w) **Total Running time** O(n + 2^m)

→ Other alternative: O(nm)

To avoid these expensive running times

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

Is small! ☺
size is only O(m)

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

W3C DTD-defin.

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)

Algorithm

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

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

n = length(w) **Total Running time** O(n + 2^m)

→ Other alternative: O(nm)

To avoid these expensive running times

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

Is small! ☺
size is only O(m)

How does `BuildFA(e)` work?
"**Glushkov automaton**" = "position automaton"
/ more details later, if time permits

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)

Algorithm

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

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

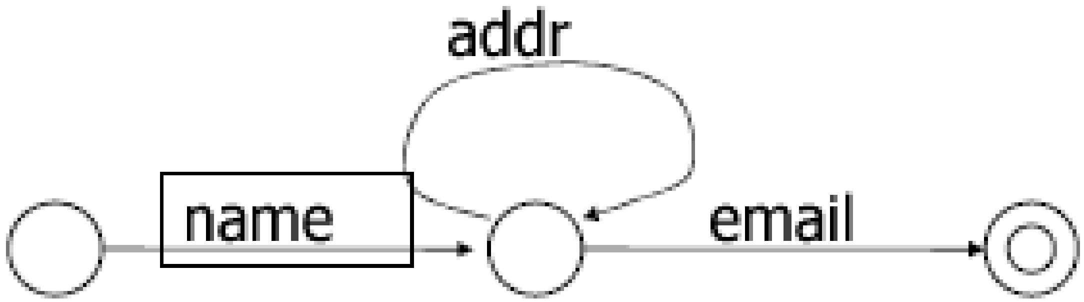
n = length(w) **Total Running time** O(n + 2^m)

→ Other alternative: O(nm)

Regular Expressions

- Each regular expression determines a corresponding *finite-state automaton*
- Let's start with a simpler example:

name, addr*, email

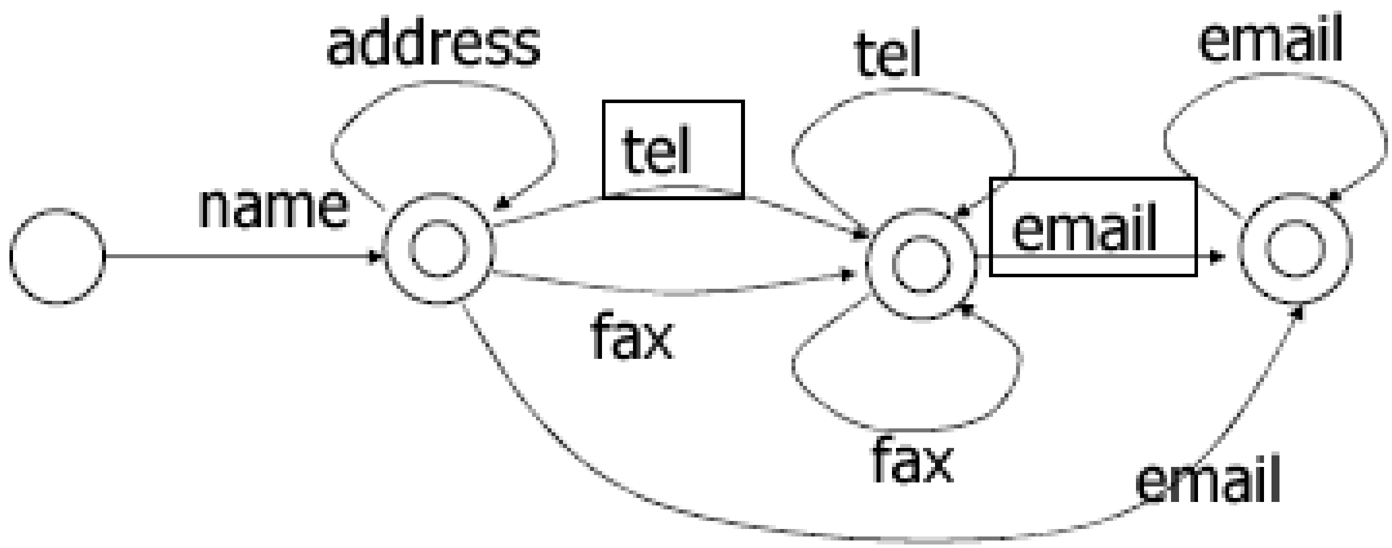


A double circle denotes an accepting state

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 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.,

<!ELEMENT ndeter

((movie|director)*,movie,(movie|director))>

This is not allowed in a DTD

(a|b)*a(a|b)

To summarize

In order to check whether a (large) document
is **valid** wrt to 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**!

To check if a Reg Expr is **allowed in a DTD**
we have to construct a particular finite automaton: the **Glushkov automaton**.

To summarize

Next, let us look at some other (minor) issues

→ Unordered lists (permutations)
→ Recursive DTDs

Some Things are Hard to Specify

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

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

Suppose that there were many more fields!

Recursive DTDs

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

What is the problem with this?

A parser does not notice it!

Recursive DTDs

```
<DOCTYPE genealogy [
   <!ELEMENT genealogy (person*)>
   <!ELEMENT person (
                  name,
                  dateOfBirth,
                  person,        -- mother
                  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 descendent of herself.

Recursive DTDs (cont'd)

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

What is now the problem with this?

Recursive DTDs (cont'd)

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

What is now the problem with this?

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

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

  ✓ For $aaa \longrightarrow$ three witnesses: $a_1a_1a_2$ $a_1a_2a_3$ $a_2a_3a_3$

  ✓ Unambiguous equivalent expression : $(a + b)^*a$

(this and next 2, from: www.infosys.uni-sb.de/teaching/streams0506/slides/stoyan.mutafchiev.slides.**ppt**

Document Type Definitions (DTDs)

- Is it enough for our purpose if the regular expression is unambiguous ? *No, it is not enough*
- the same unambiguous regular expression:

$$(a + b)^*a \xrightarrow[\text{subscripts}]{\text{mark with}} (a_1 + b_1)^*a_2$$

- consider : baa
  - ✓ one witness: $b_1a_1a_2$ (unambiguous)
  - ✓ it is not possible to decide b_1a? without looking ahead
- *Without looking beyond that symbol in the input word* [1-unambiguous]

$(a + b)^*a$ ≡ ?

unambiguous 1-unambiguous

Can you find a 1-unambiguous Reg Exp for $(a + b)^*a$
… not so easy…. ☺

Document Type Definitions (DTDs)

- Is it enough for our purpose if the regular expression is unambiguous ? *No, it is not enough*

- the same unambiguous regular expression:

$$(a + b)^*a \xrightarrow[\text{subscripts}]{\text{mark with}} (a_1 + b_1)^*a_2$$

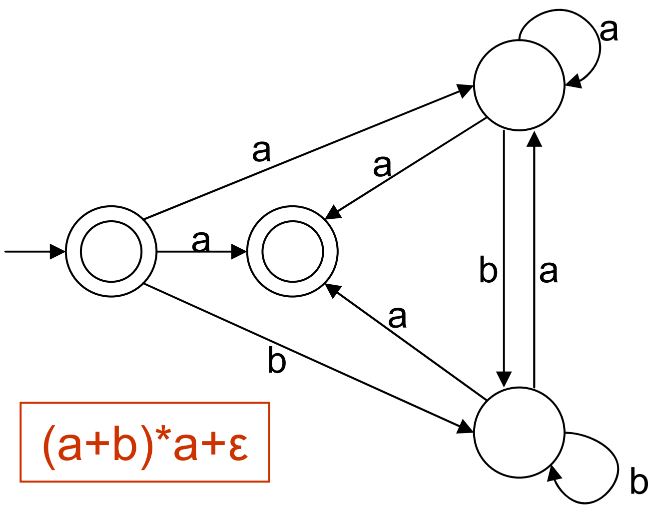
- consider : baa
  - ✓ one withness: $b_1a_1a_2$ (unambiguous)
  - ✓ it is not possible to decide $b_1a?$ without looking ahead
- *Without looking beyond that symbol in the input word* [1-unambiguous]

$$\underset{\text{unambiguous}}{(a + b)^*a} \quad \equiv \quad \underset{\text{1-unambiguous}}{b^*a(b^*a)^*}$$

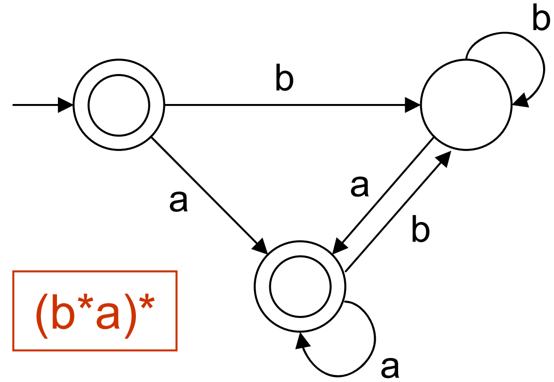
Document Type Definitions (DTDs)

[Brüggemann-Klein, Wood 1998]:

- Can we recognize deterministic regular expressions?
  - ✓ *A regular expression is deterministic (one-unambiguous) iff its Glushkov automaton is deterministic.*
  - ✓ *The Gluschkov automaton can be computed in time quadratic in the size of the regular expression*



(a+b)*a+ε



(b*a)*

Glushkov's automaton

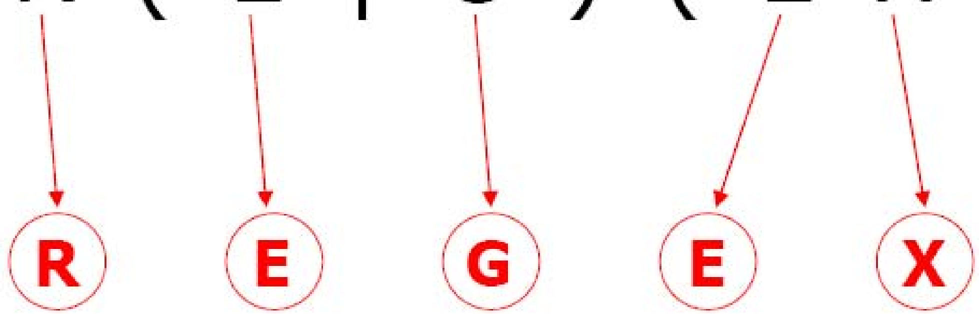
R (E | G) (E X) *

Following slides from: http://www.cs.ut.ee/~varmo/tday-rouge/tammeoja-slides.pdf

Glushkov's automaton

- Character in RE = **state** in automaton

R (E | G) (E X) *



Glushkov's automaton

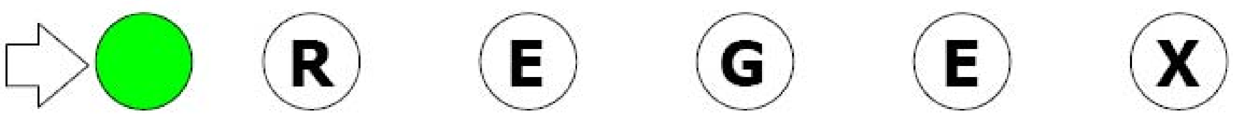
- Character in RE = **state** in automaton
  + one state for the beginning of the RE

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) *

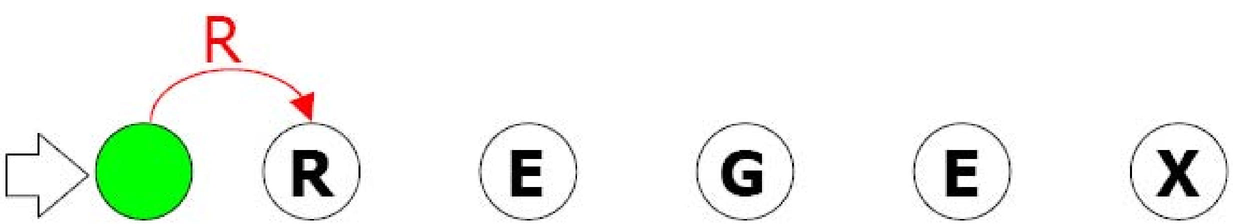


R...

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) *

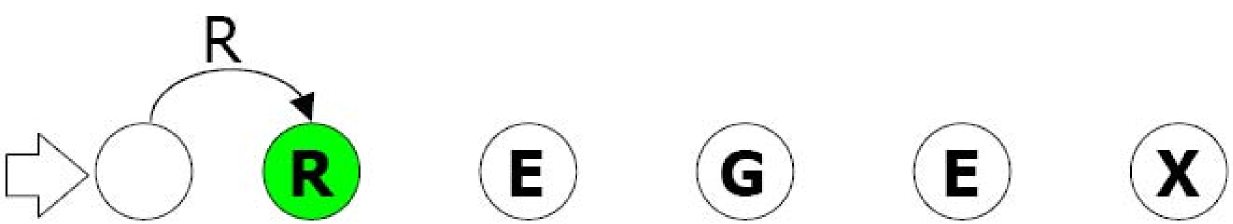


R...

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) *

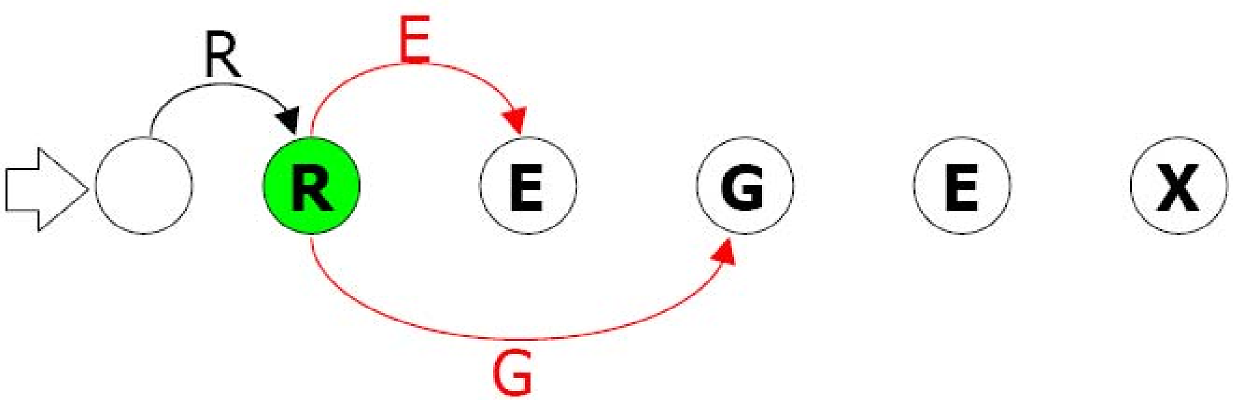


R...

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) *

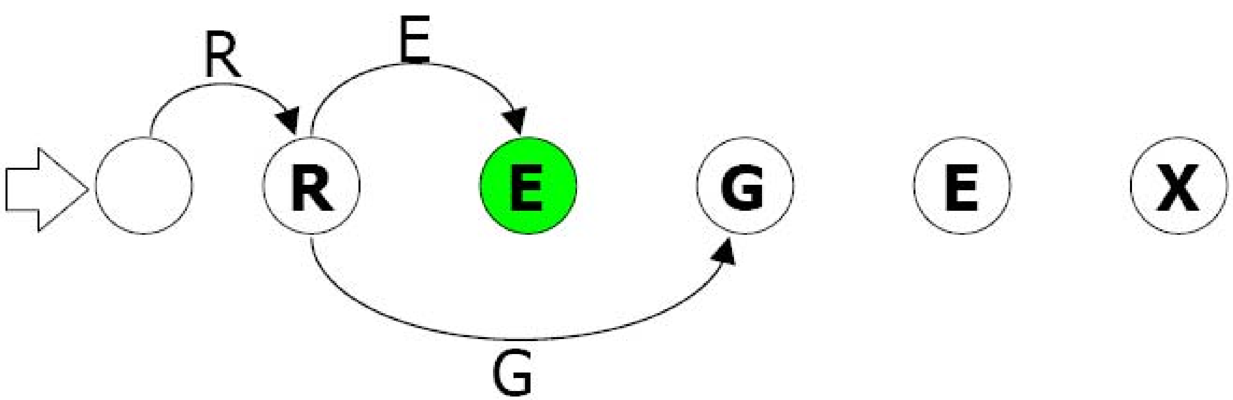


RE...
RG...

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) *

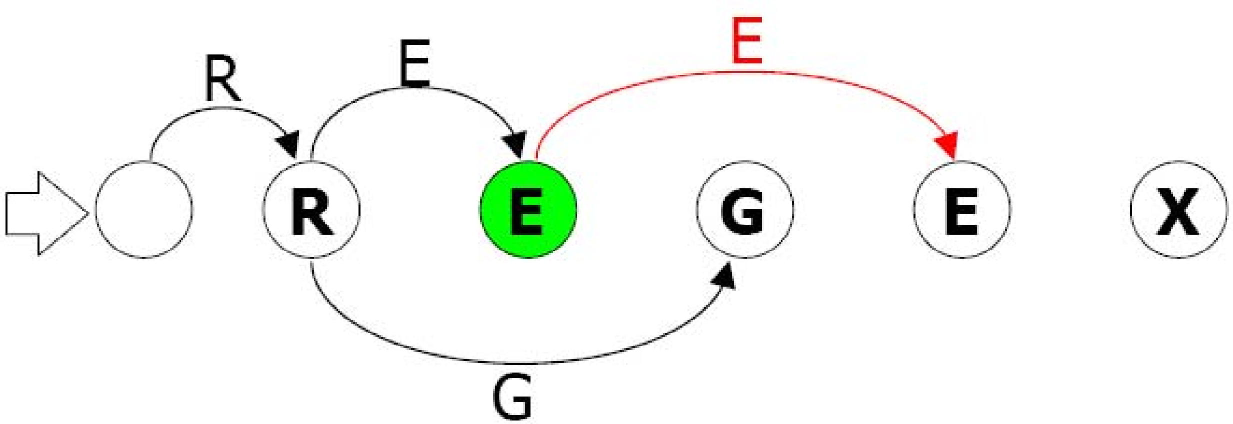


RE...

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) *

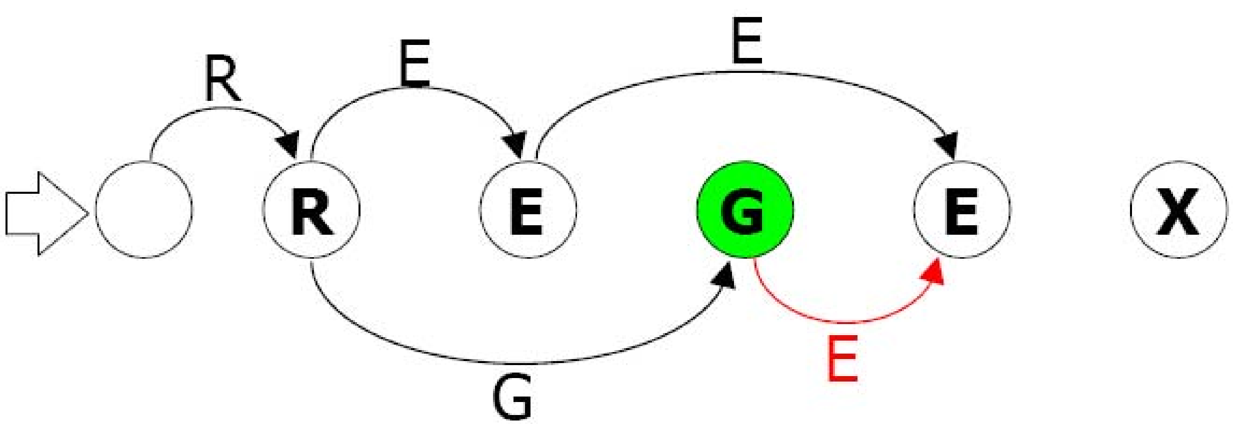


R**E**E...

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) *

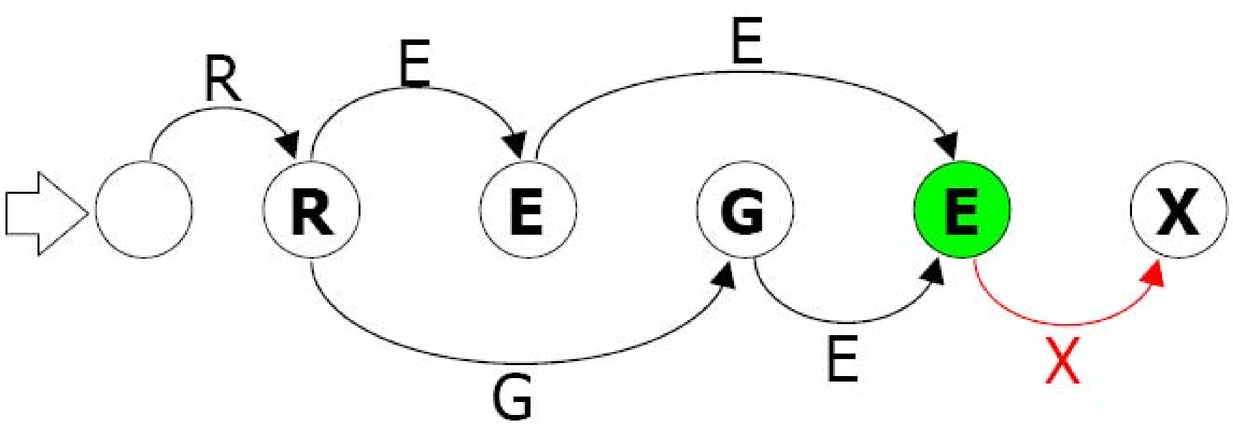


R**G**E...

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) *

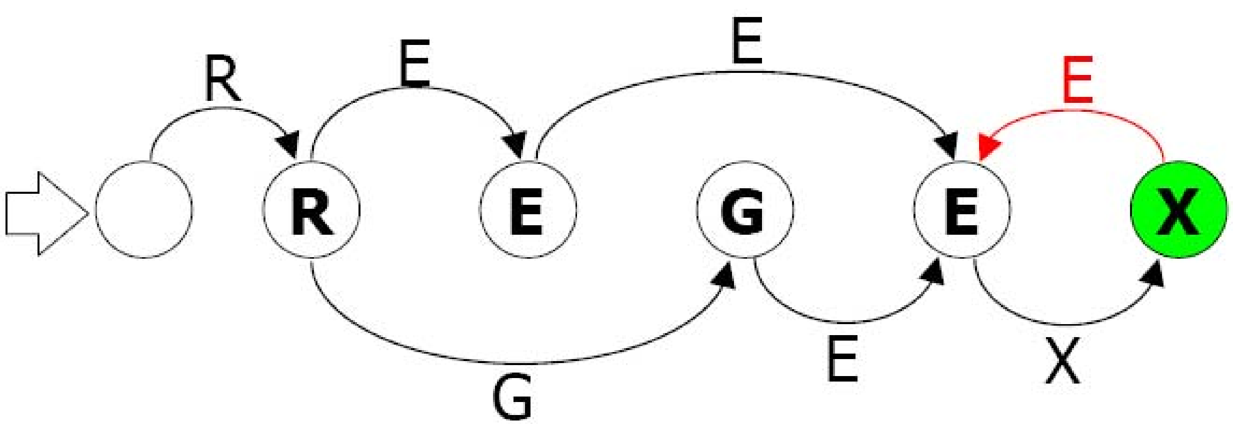


RG**E**X...

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) *

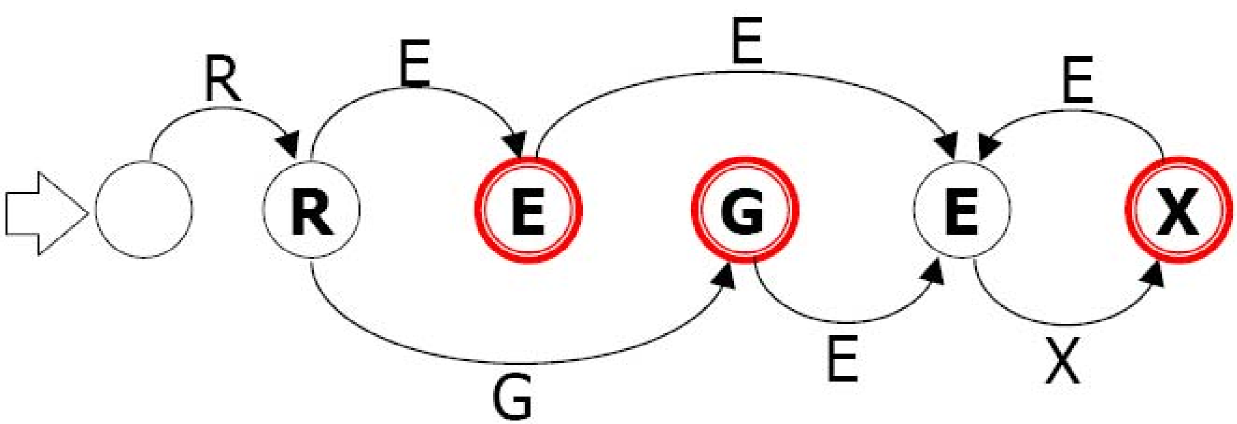


RGE**X**E...

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

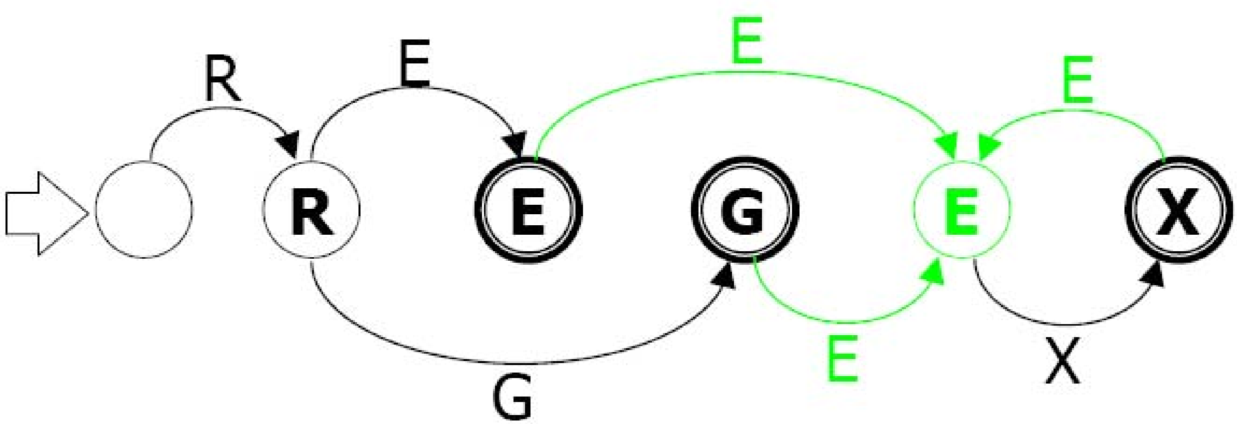
R (E | G) (E X) *



Glushkov's automaton

- All labels entering a node are labeled by the same character

  for example **after reading character 'E' only states with label 'E' can be active**



Questions

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)

Questions

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)

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

How many transitions are in the Glushkov automaton for F?

And how many in F's minimal automaton?

Does F contain: $v = a_3 \, a_2 \, c$

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.

Given an input string w of length m, it takes us time
$O(n^2 + m)$ to check w against E.

Can this be improved for the case the m is small (non-quadratic)
with resepect to n?

→ do not want to construct the full automaton, because
that is too expensive..

END
Lecture 4